2013 IEEE 37th Annual Computer Software and Applications Conference

A Software Architecture for High-Level Development of Component-Based Distributed Virtual Reality Systems

Vasily Y. Kharitonov
Department of Computers, Systems and Networks
National Research University "Moscow Power Engineering Institute"
Moscow, Russia
e-mail: kharitonovvy@gmail.com

***Abstract*—Distributed virtual reality systems (DVR systems) have evolved significantly over the past twenty years and found wide spread in many applications for training, educational and entertainment purposes. Modern DVR systems require sophisticated data exchange mechanisms to provide consistent interaction of a large number of users over the Internet. While many of these mechanisms have been well studied, usually they represent isolated solutions requiring knowledge of low-level networking programming for implementation. In this regard, there is still a lack of universal, easily deployable and extensible software architecture that enables rapid creation of complete systems from scratch. In this paper we present architecture of a middleware allowing an application developer to easily implement and deploy custom DVR systems for specific tasks without direct low-level network programming.**

Keywords-software architecture design; middleware; high-level development; virtual reality; distributed simulation

I. INTRODUCTION

At present, with the growth of computational power of modern computer systems and intensive development of networking technology distributed virtual reality systems (DVR systems) have evolved considerably and proved its effectiveness and applicability in many application areas such as science, education, engineering, medicine and entertainment. Such systems consist of many geographically separated *components* interacting each other in real-time with intent to create a common *virtual environment* (VE) which can be shared between multiple participants. Depending on the DVR system type, components can perform various tasks. For example, component may represent a separate computer participating in online computer game, or a complex system computing physical models of vehicles in modern battlefield simulations. Regardless of the system type, consistent interaction of DVR system components should be provided.

Development of a DVR system is a complex and challenging process which requires both good programming skills and deep knowledge of distributed systems theory. While many implementations of DVR systems have been developed in the past decades there is still a lack of top-down approach to build the system of such kind.

The main aim of this work is to develop software architecture providing means for high-level development of a DVR system from scratch. To do this we should answer several questions. First, what mechanisms should be built into a DVR system to ensure a consistent interaction of a large number of components? Second, how can we overcome physical limitations affecting data consistency and scalability in DVR systems, like latency and unreliable data channels? And third, since from the software point of view a DVR system is a complex, real-time distributed application, how to make its development easier for the end user? In this paper we will try to answer all these questions.

The remainder of this paper is structured as follows: Section II gives a brief overview of related work in area of DVR systems. Section III describes the proposed software architecture. Section IV introduces the *Distributed Virtual Reality Protocol* which is a basis for process interaction within the architecture. In Section V consistency maintenance mechanisms implemented in the proposed architecture are discussed. The process of a DVR system high-level development and the proposed middleware implementation details are considered in Section VI followed by a number of examples showing how to work with provided API. Section VII describes performance evaluation of the proposed framework. In Section VIII further ways to improve scalability of the proposed architecture are discussed. We conclude the paper with applications and future work in Section IX.

II. RELATED WORK

Modern DVR systems can be classified according to the application area into three main branches: simulation systems, academic research projects and multiplayer computer games [1].

The first type of systems is mainly used in various military simulations: from aircraft simulators to large-scale battlefield simulations. The well-known representatives are SIMNET and its followers that had become industry standards – DIS (IEEE Standard 1278) and HLA (IEEE Standard 1516) [2].

Projects for research and academia are mainly focused on improving experience of interactive multi-user collaboration in areas requiring joint work of many participants like distance learning, virtual conferencing and distributed modeling. Examples of such systems include: DIVE, RING, NPSNET [3], MASSIVE [4], MR Toolkit, Avango [5].

Multiplayer computer games are the most widespread type of DVR systems and use similar with others mechanisms. Among the most successful game series there are *First-Person Shooter* (FPS) games like *Unreal Tournament*, *Counter Strike: Source*, and *Quake*. Nowadays there is a growing interest in *Massively Multiplayer Online Games* (MMOGs) and *metaverses* [6], which support especially large numbers of users (currently, up to hundreds of thousands).

The closest analogue of the proposed framework among existing DVR systems is HLA. The High Level Architecture for Modelling and Simulation was developed to facilitate interoperability among simulations and promote reuse of simulation components. Using HLA, computer simulations can communicate to each other regardless of the computing platforms. Communication between simulations is managed by a *Run-Time Infrastructure* (RTI), which can be thought as DVR middleware. However, the HLA standard only defines RTI specification but not its implementation, resulting in that heterogeneous RTIs may not interoperate properly. In addition, various RTIs provide different performance which in most cases insufficient for simulating VE state at high rates. Moreover, HLA supports only peer-to-peer communication architecture which does not scale up well for large scale geographically distributed simulations and can be effectively used only on LAN network.

Summarizing all the systems above we may notice that while many DVR systems have been implemented in previous decades there is still a lack of universal, easily programmable and extensible framework (middleware) allowing not experienced user to create a new DVR system in a short time. Almost all the systems considered are proprietary and there is no standardized mechanism and methodology describing how a DVR system can be developed from scratch. Meanwhile, development of a DVR system is quite complicated process requiring both good programming skills and deep knowledge of distributed systems theory.

In this regard we propose a solution designed to simplify a DVR system development process. By encapsulating all the low-level mechanisms inside the framework we have created a DVR system middleware, providing easy and user-friendly API to create and manage the shared virtual environment.

III. Proposed Software Architecture

A. Basic assumptions

From the software point of view, any DVR system can be represented as a collection of independent processes, interacting with each other over the communication network.

Without loss of generality assume that every process runs on a separate computer node. There is no shared memory between processes. All communications are carried out only through the message exchange between processes using a *high-level protocol.* Also processes do not share global clock that is instantaneously accessible to them. Processes execution and messaging are asynchronous: processes may execute actions at arbitrary time instants and sending messages is not blocking.

B. Communication Architecture

Process interaction is based on a certain type of *communication architecture*, which determines the topology of process interconnection and process roles in the data exchange. The main types of communication architectures are *client/server, peer-to-peer* and *multi-server.*

In this work we mostly oriented to development of medium-scale DVR systems (up to 100 users) so client-server communication architecture ideally fits to our needs. However, we extend standard client-server architecture by supporting IP multicasting between clients. Thus, along with sending data through the server, direct client communication is also possible. Multicasting allows to send data to only those processes who are interested in their receiving. In other words, processes are able to *subscribe* on data. As a result, server workload as well as data delivery time can be significantly reduced.

C. The Software Architecture

Fig. 1 depicts the main parts of the proposed architecture. According to the picture, a DVR system is considered as a set of processes (shown as large bars) interacting with each other using a certain type of communication architecture (client/server in our case). Each process contains a number of units and represents a separate component of a DVR system.

The DVR system component encapsulates all the task specific functionality. This is a customizable part of the software working on top of the proposed middleware. User is able to process his own input devices and attach specific visualization system to render VE state.

Each DVR component depending on the specific task may be either passive or active. Passive components are just the observers of current VE state and can be connected to the DVR system at any moment. Active components are able to modify the VE state by creating new objects or changing states of existing ones.

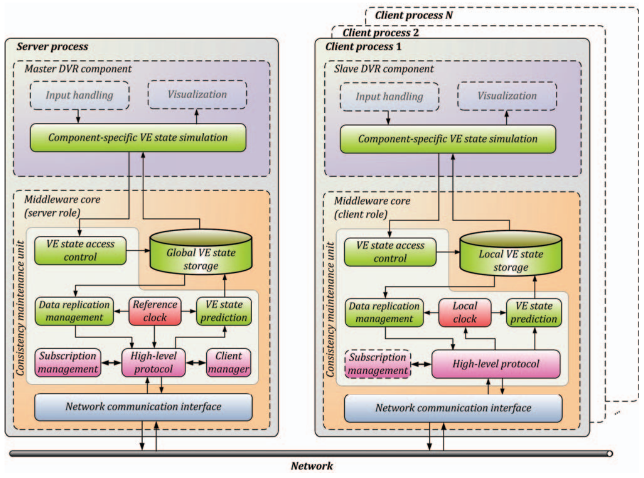


Figure 1. Proposed software architecture

According to the client/server architecture, master and slave components also can be distinguished. Master component is located on the server side and is generally used to coordinate slave components, providing their interaction in the virtual environment, and to monitor the state of the entire system.

DVR component consists of three units. *Input handling* unit is optional and is only present in active components. It is used to read data from user input devices and converts them to control actions that are transmitted to the *Component-specific VE state simulation* unit which simulates current VE state for the component. *Visualization* unit is also optional and can be presented in both active and passive components. It is used to render current VE state for the component.

Each client process includes *Local VE state storage* where it keeps the part of the VE state needed to visualize view within component visibility scope. *VE state access control* unit controls access to the storage checking whether current process is permitted to modify VE state.

Global VE state storage, located on the server, keeps a reference copy of the entire VE state. All changes performed by processes in their local storages are automatically replicated to the global storage. When needed, the data not presented in the local storage are loaded from the global storage. This happens, for example, when a new client connects to the system or when new VE objects are created. There is a bidirectional data exchange between global storage on the server and local storages on the clients all the time. Together server and client storages form *distributed data storage*.

It is convenient to organize information in the data storage in a hierarchical structure which could be shared between processes, called *distributed scene graph* (DSG) [7]. DSG can be described as a directed acyclic graph $G = (V,E)$, where V – set of vertices, each corresponds to a separate VE object or group of objects, E – set of edges establishing logical and spatial relationships between the VE objects.

The most important unit in each process is *Consistency maintenance* unit. It resolves the following issues:

- interaction of current process with other processes (using *High-level protocol* and *Network communication interface*);
- initialization of new clients (*Client manager*);
- data replication management, subscription management, clock synchronization;
- hardware limitations compensation using dead reckoning technique (*VE state prediction* unit, see section V);
- concurrency control, object ownership management (*VE state access control* unit).

IV. High-Level Protocol

Process interaction within the proposed architecture is based on a high-level protocol, named *Distributed Virtual Reality Protocol* (DVRP). DVRP operates on top of two transport protocols: TCP and UDP, combining reliability features of the former and speed of the latter. Individual messages that require reliable delivery are transferred via TCP (e.g. messages on VE objects creation). For frequently transmitted messages that require fast delivery and are not critical for packet loss UDP is used. Such combination of protocols provides a more efficient use of bandwidth.

A. Protocol Message Groups and Format

For ease of further protocol expansion its messages are divided into several groups:

- *world messages* – global operations on VE state modification;
- *objects messages* – individual objects manipulation: objects creating/deleting, scene graph modification, object ownership operations;
- *camera messages* – inform remote users on current user's position and orientation within VE;
- *attribute messages* – object state attributes change;
- *service messages* – subscription services, collecting of network statistics, process interactions, *Remote Procedure Call* (RPC) implementation etc.

One of the distinguishing features of the proposed protocol is the ability to use variable-length messages. This is achieved by applying bit streams instead of predefined protocol message structures (see Fig. 2.a). The only predefined field in the message is its header. Header specifies the group which message belongs to (*SuperID*) and the position inside this group (*ID*). Each message is a bit stream, which is assembled on the fly. Message writing starts from the header followed by specific for concrete message type fields.

Protocol messages handling on the receiving side is made in the same order as the assembling when sending. After extracting message header, appropriate handler is retrieved and if it exists, message remainder comes to its input where it is handled. Using bit streams makes it possible to create multipurpose handlers which are able to change the processing logic depending on the message contents.

Bit streams make it possible to combine multiple messages into a single one (see Fig. 2.b). For this purpose the protocol has *batch* message type. After writing batch message header into stream, compound messages are written sequentially. Another bit stream application is VE object serialization. At each simulation cycle state attributes of all modified objects are serialized one by one into a single bit stream and sent to the network for a single *Send()* call.

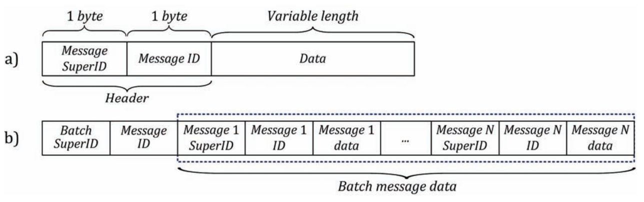


Figure 2. The formats of DVRP protocol messages

B. Types of Process Interaction

There are two main types of process interaction in DVRP: *client-server* and *client-client.*

1) Client-server interaction: a basic type of interaction and corresponds to the selected client-server communication architecture. Its main phases are shown in Fig. 3.

a) Connection phase: describes a new client connection to the DVR system. When client starts it establishes three connections to the server: basic TCP-connection, RPC TCP-connection and UDP-connection (interaction types 1, 2, 4). Basic TCP-connection is used for data requiring reliable delivery. RPC TCP-connection is used for blocking RPC-requests. Connection over UDP-protocol, as mentioned before, is used to deliver data which are not sensitive to losses.

Once all three connections are successfully established, server assigns a unique identifier to a client used to identify it in the system (interaction type 5). Server then initializes the client by sending him service information, including information on already connected clients (interaction type 6), as well as data about the current VE state (interaction type 7). In response, client synchronizes its clock with server clock (interaction type 8) (see Section V). Starting from this moment client is considered connected to the DVR system.

b) Interaction phase: describes possible client-server interactions after client connection. The main actions that clients can undertake in this phase include new objects creation, ownership operations and existing objects states modification (interaction types 9, 11, 12). Server informs other clients when VE state modification occurs (interaction types 10, 13). Objects state changes can be delivered both over UDP and TCP, depending on the selected for the state attributes transport protocol.

c) Disconnection phase: The client is considered disconnected, if it closes TCP-connection, or unexpected disconnection occurs, for example, caused by a network failure (interaction type 14). In any case, server removes disconnected client from the client list and notifies other clients (interaction type 15).

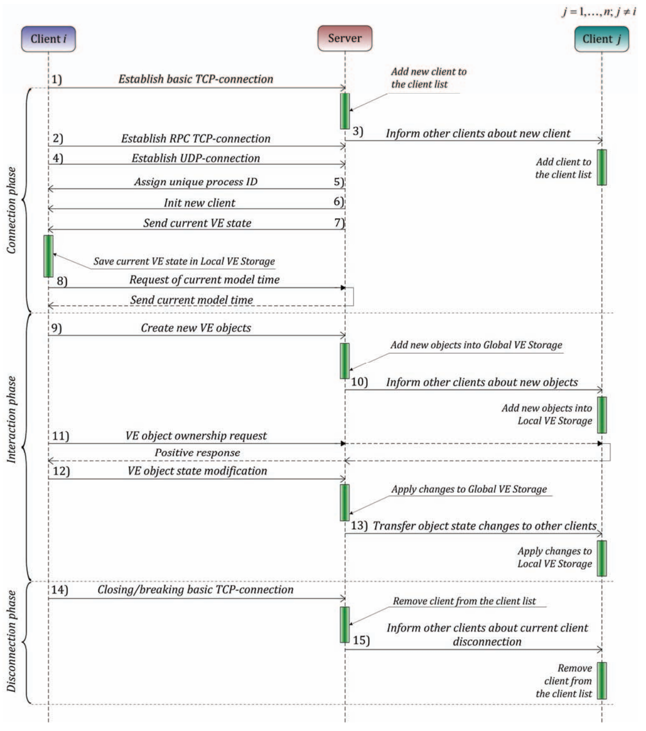

$j = 1,\ldots,n;\ j \neq i$
Client i
Server
Client j
1) Establish basic TCP-connection
Add new client to the client list
3) Inform other clients about new client
2) Establish RPC TCP-connection
4) Establish UDP-connection
Add client to the client list
Assign unique process ID 5)
Init new client 6)
Send current VE state 7)
Save current VE state in Local VE Storage
8) Request of current model time
Send current model time
9) Create new VE objects
Add new objects into Global VE Storage
10) Inform other clients about new objects
Add new objects into Local VE Storage
11) VE object ownership request
Positive response
12) VE object state modification
Apply changes to Global VE Storage
13) Transfer object state changes to other clients
Apply changes to Local VE Storage
14) Closing/breaking basic TCP-connection
Remove client from the client list
Inform other clients about current client disconnection
15)
Remove client from the client list
Connection phase
Interaction phase
Disconnection phase

Figure 3. Interaction phases between client and server in DVRP

2) Client-to-client interaction: can be carried out either through the server, or directly between clients when using multicast. In the first case client uses a special address type message. When server receives this message it retranslates message contents to the corresponding client. As a result, for the client message transferring looks transparent as if it sends the message directly to the designated client.

In the second case, client-to-client interaction assumes that client sends data to the specific multicast group, which other clients are subscribed to. In this case, data delivery is performed by the underlying transport protocol. Note that direct work with multicast groups is hidden from the user. These groups are automatically created when adding new VE objects or registering state patterns (see Section VI).

V. Consistency Maintenance Mechanisms

Proposed architecture provides a number of mechanisms for consistency maintenance within a DVR system, namely (see Section III): *data replication management*, *clock synchronization*, *VE state prediction* and *concurrency control*. All these mechanisms operate on top of DVR protocol and together provide instruments to control data consistency over the system.

A. Data Replication Management

As interaction of components in a DVR system is carried out in real time, update and, in particular, rendering of the VE state should be performed at high rates (e.g. in military simulations – 60Hz). Since all the data between processes transmitted over the network, the transfer of complete objects states with such frequency would require a very high bandwidth which is unacceptable when using WAN network.

A complete object state transfer is often redundant, and needed only on the side of the process directly managing the object state. In most cases for the rest of the processes there is no need to simulate accurate object state and a simplified state model can be used. The main idea of the *selective consistency principle* is to distinguish the most essential for the specific task object state attributes and to maintain consistency explicitly on them. These attributes can be selected on the basis of different assumptions. The most important of them is human perception. The selective consistency principle is a relaxation of absolute consistency notion and supposes that there can be inconsistencies in objects states but they are negligible for the user.

The main features of selective consistency principle are:

a) dynamic selection of VE objects state attributes to be replicated;

b) strategy-based object state replication management;

c) use of delta compression when serializing objects;

d) use of simplified VE objects models on the processes that are not owners of objects states; it involves state prediction techniques, such as dead reckoning [1].

Let us consider a) – c) positions in more detail. Position d) will be covered in the corresponding section.

High-level protocol provides the user a flexible mechanism to manage object state replication. This mechanism is based on assigning one of four replication strategies to the state attribute:

- *manual replication* (*MR strategy*) – attribute is replicated only when method *Serialize()* is called;
- *replication on every change* (*REC strategy*) – changing the attribute value at any process causes it immediate replication to other system processes;
- *fixed rate replication* (*FRR strategy*) – attribute value is replicated with a predefined rate;
- *conditional replication* (*CR strategy*) – attribute replication is determined by the user-defined logical expression (condition).

The above strategies provide more efficient use of network bandwidth. Among all of them, REC strategy ensures better consistency, but at the same time, it also produces higher traffic. FRR strategy provides acceptable consistency and more uniform bandwidth utilization due to the fact that not all attribute value changes are transmitted over the network. The two remaining strategies (MR and CR strategies) are the most flexible and allow the user to implement his own replication algorithms.

Delta-compression (position (c) of the principle) means that in each simulation cycle only changed parts of VE objects states are passed over the network. Using delta-compression is obvious and at the same time is extremely beneficial because it allows a DVR system to significantly reduce consumed network bandwidth without any loss in consistency. It works as follows. When attribute value is changed, in accordance with the selected replication strategy it can be marked as requiring replication. All attributes of the object, marked in current simulation cycle are packaged into a single message and sent over the network.

B. *Clock Synchronization*

Clock synchronization is the technique of ensuring that physically distributed processes have a common notion of time. Clock synchronization assumes availability of a reference clock in the system that is stored on a dedicated process called time server. In our architecture, each newly connected client synchronizes its clock with the server clock using a special procedure, similar to Cristian algorithm [8] and Network Time Protocol [9].

Clock synchronization is very important when building a DVR system. Reference clock set the course of the internal model time which should be common for the entire system. Each global VE state is calculated based on the model time and all events in the VE are related to particular instants of model time.

The proposed synchronization method involves computation of two values:

- $\hat{d}$ – estimates *round-trip time* (*RTT*);
- $\hat{q}$ – estimates the server-client clock offset.

The basic step of the method is shown in Fig. 4. Suppose that at some moment of time τ shift between server and client clocks equals to q_i. Synchronization procedure works as follows. At time $T_{i-3}^{Cl} = C_{Cl}(t_{i-3})$ the client sends a synchronization request to the server (see message *m* at Fig. 4). After receiving request server processes it and responds at time $T_{i-1}^{S} = C_{Cl}(t_{i-1})$ with the message m' containing server times T_{i-1}^{S} (message *m* receipt time) and T_{i-2}^{S} (message m' send time). Client receives server message m' at time T_i^{Cl}. Finally, estimations $\hat{q}_i$ and $\hat{d}_i$ can be calculated based on information in the message m' using the following expressions (see [9] for more information):

$$\begin{gathered} \hat{q}_i = \frac{T_{i-2}^{S} - T_{i-3}^{Cl} + T_{i-1}^{S} - T_i^{Cl}}{2}, \\ \hat{d}_i = T_{i-2}^{S} - T_{i-3}^{Cl} + T_i^{Cl} - T_{i-1}^{S} \end{gathered} \tag{1}$$

To improve the accuracy of clock offset computation, in our approach we make 8 iterations. At each iteration pair $(\hat{d}_i, \hat{q}_i)$ is stored. After all the iterations the value $\hat{q}$, corresponding to the minimum RTT $d_{\min} = \min(\hat{d}_i)$ is taken as the true clock offset value. Based on $\hat{q}$, the clocks on the client side are adjusted.

It can be shown that the absolute error of the considered method is $\pm d_{\min}/2$, which means that the synchronization accuracy is close to the minimum value of the latency.

C. *Object State Prediction*

The idea behind VE object state prediction, also known as *dead reckoning* technique, is to transmit object state update messages less frequently, while calculating the missing states by adding extra information into the messages. As a result, it becomes possible to reduce the network traffic and decrease the latency.

One of the most common approaches to predict object state is the use of extrapolating polynomials derived from the Taylor expansion of position function $\boldsymbol{r}(t)$. According to Taylor formulae, for a given position function $\boldsymbol{r}(t)$ and its ε-neighborhood $U(t,\varepsilon)$ the position of object at some moment of time $t + \Delta t \in U(t,\varepsilon)$ can be calculated as follows:

$$r(t+\Delta t) = \sum_{k=1}^{\infty} \frac{r^{(k)}(t)}{k!} \Delta t^k \tag{2}$$

where $r^{(k)}$ is a derivative of $\boldsymbol{r}(t)$ of order *k*.

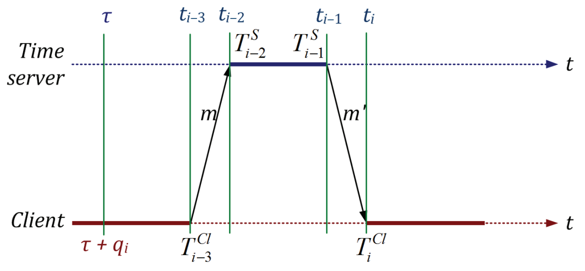


Figure 4. Basic step of the proposed clock syncronization method

Most often the Taylor series expansion is limited by the first or second derivatives. These derivatives characterize, respectively, the velocity and acceleration of the object.

The use of dead reckoning involves consideration of two issues: the choice of update rate at the local side and the choice of prediction algorithm at the remote side. These issues are interrelated. Low update rate leads to a lack of prediction accuracy. Conversely, too high update rate results in an inefficient use of network bandwidth. Therefore, update rate should be chosen in a way that ensures a balance between consistency and consumed network traffic for a specific prediction algorithm.

While many dead reckoning algorithms have been implemented (for reference, see [10,11]) in the past decades there is still a lot of challenges regarding choosing an optimal update rate and accurate prediction technique.

Within the proposed architecture we have implemented a novel *motion-aware adaptive dead reckoning algorithm* [11]. Our approach is different from the algorithms implemented in previous works in that it recognizes object motion patterns and makes use of several prediction techniques combined together (see Fig. 5).

As a result, better prediction accuracy for objects moving along complex trajectories can be achieved. Moreover, it is adaptive in the sense it adjusts update rate and the size of update messages according to the motion pattern, which, in turn, enables more flexible use of network bandwidth.

During the performance evaluation, the proposed algorithm demonstrated very good performance compared to existing approaches providing the same level of accuracy with much less network traffic consumed [11].

D. Concurrency Control

When building a DVR system it is very important to correctly handle requests of multiple components to the same data. This situation often occurs, for example, when multiple components attempt to simultaneously manipulate the same object. The actions of different components overlapping in time should not interfere with each other. To achieve this, a mechanism is required to control access to the objects states.

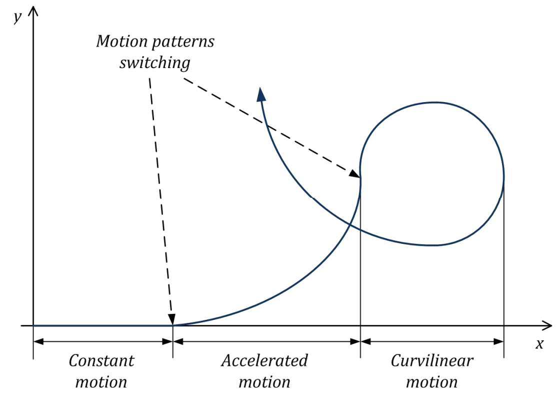


Figure 5. Motion-aware approach applied to a complex trajectory

The proposed architecture uses an approach based on object ownership transferring. Each VE object can be either free or owned by a single process. This ownership can be transferred between processes. Any other process may try to seize the object by sending corresponding request to the server process. If the requested object is free (i.e. not occupied by another process), then the server marks it in a global storage as "owned" and sends the requesting process a positive response. After this only owner process is able to alter object state in the global storage. If the requested object is owned by another process, the server sends ownership request to this process. The owner process may either meet this request and grant membership or reject request, depending on his needs. Owner process may also block access to the object. In this case any membership request is automatically declined.

VI. The TerraNet Framework

The architecture, high-level protocol and associated consistency maintenance mechanisms proposed in this paper have become the basis for the software framework for high-level development of DVR systems. The main goal when designing the TerraNet framework was to create a powerful tool which enables developer to work with a DVR system at the user level of abstraction (i.e. the level of VE objects) while hiding from him all low-level networking mechanisms.

That is why we decided to develop our software as a middleware with the appropriate high-level API. Nowadays middleware development is considered to be one of the most promising areas of DVR research [12]. DVR middleware is a layer of software between operating system with its low-level networking APIs and user application, providing transparent interaction of the application with other DVR components.

A. TerraNet Framework Overview

The TerraNet framework includes several libraries:

- *terranet_core* – core library with all the functionality needed to deploy the DVR system infrastructure;
- *terranet_osg* – graphic library including built-in window system and scene graph rendering system;
- *network* – networking library implementing basic network functionality such as creating network connections, data transferring, multicasting etc.

Framework structure and relationship between its parts, user application and operating system (OS) are shown in Fig. 6.

The main features of the TerraNet framework:

- manipulating with VE objects at the level of state attributes, support for hierarchical relationships between VE objects (distributed scene graph);
- subscription-based data replication mechanisms;
- ability to assign user-defined *controllers* to modify default objects behavior;
- object ownership management (concurrency control);
- core library is separated from the graphical components which allows the developer to use this library together with any visualization system;
- portability: all code written in standard C++ language, using cross-platform libraries including *Sockets*, *OpenGL* and *OpenSceneGraph* [13].

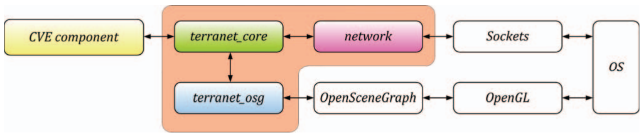


Figure 6. The TerraNet framework structure

B. *TerraNet Core API Classes*

TerraNet core API classes are based on the well-known *Model-View-Controller* (MVC) concept [14] which identifies three separate components of the application: *data model*, *graphical representation* and *control logic*, so modification of one component has minimal impact on others.

In the TerraNet core API model component is represented by interface classes *TA_World*, *TA_Object*, *TA_State* and *TA_StateAttribute. TA_World* class is a container storing the whole scene graph, *TA_Object* class describes a single scene graph node associated with a particular VE object. *TA_State* and *TA_StateAttribute* classes are used to work with VE object state. User can create/delete attributes of various types, change their values, define replication strategies etc.

Control logic component is represented by *TA_Controller* class and its derived subclasses. Controller allows the user to define his own rule of object state changing. For example, controller may implement dynamics of a jet fighter. In a more general sense, controller can be used to describe any algorithm for object state processing.

TerraNet core API provides a centralized mechanism of resource management through *TA_ResourceManager* class. Almost all the resources can be allocated using resource manager which is a singleton object [14] and accessible from anywhere in the DVR application. User does not need to think about deleting resources because all of them are automatically deleted upon completion of the program.

C. *Creating Applications with TerraNet Core API*

Writing application using TerraNet API begins with an initialization. First, current process role in the communication architecture (client or server) is defined. Next, network and graphical components are started, scene graph is created and clocks are synchronized with the server. The following code fragment shows an example of client application initialization:

```
// Get an instance of Resource manager
TA_ResourceManager* pResMan = TA_ResourceManager::GetInstance();
// Init current process as a client application and tell
// the core to use OpenSceneGraph for rendering
pResMan->Init(TA_CLIENT, new OSGResourceManagerImpl);
// Get a network interface
TA_NetworkInterface* pNetIntrf = pResMan->GetNetworkInterface();
// Connect our process to the server
if (!pNetIntrf->Connect("192.168.1.1", true))
      return false;
// Perform clock syncronization
pResMan->GetTimer()->Sync();
// Get an instance of the virtual environment
TA_World* pWorld = pResMan->GetWorld();
// Create windows and cameras (optionally) ...
// Create renderer (optionally)
pRenderer = pResMan->GetRenderer();
// Tell the renderer what to render
pRenderer->SetSceneData(pWorld);
// Get an instance of the Input Manager
TA_InputManager* pInputManager = pResMan->GetInputManager();
```

After calling network interface *Connect()* method in accordance with the DVRP protocol server generates a unique client ID and passes it to the client along with current VE state. Clock synchronization completes new application initialization. The only thing left is to run the application main loop, which looks as follows:

```
while (pInputManager->GetKey() != TA_KEY_ESCAPE)
{     // Treat current VE state
      pWorld->Simulate();
      pRenderer->RenderFrame();
}
```

D. *Operations on Virtual Environment*

Among the basic operations provided by the TerraNet API, the most important are operations on the VE state modification, including objects creating/deleting, linking objects to each other in a scene graph and objects states modification.

Creation of object assumes its registration in the global VE state storage which means assigning a unique object identifier and the name used to identify the object in all DVR processes. To associate a graphical representation with the object, the user should create a model *TA_Model* and assign it to the object. For example, to create an “Airplane” object user should write the following code snippet:

```
TA_Object* pAirplane = pResMan->CreateObject("Airplane");
pAirplane->SetModel(pResMan->CreateModel("aiplane.3ds"));
```

Linking objects in a scene graph is performed using the parent-child relationship. To add a child to the object, *AddChild()* method should be called with a child specified. Each object can have an arbitrary number of children.

Object state modification is made through *TA_State* interface class. State modification means adding/removing of attributes and their values changing. The following example shows how to do this. Suppose we need to add an “Airfield” object and then to place previously created “Airplane” object to a specific position on top of the airfield. To do this, one should add the following code snippet:

```
// Create "Airfield" object
TA_Object* pAirfield = pResMan->CreateObject("Airfield");
pAirfield->SetModel(pResMan->CreateModel("airfield.3ds"));
// Add "Airplane" object to the "Airfield" object as a child
pAirfield->AddChild(pAirplane);
// Add "Airfield" object into the root of the scene graph
pWorld->GetRootObject()->AddChild(pAirfield);
// Modify the state of the "Airplane" object
TA_State* pState = pAirplane->GetState();
TA_StateAttribute* pAttr = pState->GetAttribute("POSITION");
pAttr->SetValue(TA_Point3D(20.0, 30.0, 0.0));
```

Note that considered examples do not contain any network operations. When designing a virtual environment with the TerraNet framework user deals only with objects. The process of the scene graph creation will be automatically transmitted to all other clients connected to the system and all remote users will be able to see created objects.

E. *State Updates Subscription Mechanism*

As it was mentioned before, replication of objects states in the TerraNet framework is based on subscription mechanism. By default, any client process receives updates from all objects existing in a virtual environment. However, each process is able to subscribe on only interested objects or objects belonging to the interested types (we call these types *state patterns*). For this purpose *TA_NetworkInterface* class has two methods: *SubscribeObject()* and *SubscribePattern()*.

Originally, subscription is implemented through the creation/deletion of certain multicast groups. Nevertheless, not all clients may support multicasting technology. For such clients a special subscription emulation mode is implemented at the server, so they can also work with subscription services.

F. *User-Defined Process Interaction*

In addition to the internal mechanisms of process interaction described by a high-level protocol and hidden from the user, TerraNet core API provides a mechanism to implement custom algorithms for process interaction. The user has the opportunity to register and distribute *interactions*, which are messages of a special type with an arbitrary set of parameters. Furthermore, user can write his own handlers to process incoming interactions. Thus, any inter-process communication protocol can be implemented. Subscription services are also available for interactions. DVR processes can subscribe only to those interactions they are interested in.

VII. PERFORMANCE EVALUATION

The main aim of the TerraNet framework performance evaluation was scalability analysis. The idea behind it was to find the maximum number of components which current version of the framework can handle simultaneously without noticeable loss in consistency. For this purpose consistency metric was introduced and a number of experiments showing impact of server load on consistency were conducted.

A. *Proposed Consistency Metric*

In [15] it was shown that consistency can be defined as a spatial metric expressing difference between objects positions viewed at various DVR processes. The corresponding term, *view consistency*, was introduced. To analyze data consistency the following metric γ, called the *mean distance between trajectories*, was introduced:

$$\gamma = \frac{1}{N}\sum_{i=1}^{N}\left|r_L(t_i) - r_R(t_i)\right| \qquad (3)$$

where $r_L(t_i)$ – true (local) position of the virtual object at time t_i, $r_R(t_i)$ – predicted (remote) object position measured at remote side at the same moment of time, N – total number of measures.

It is convenient to measure the distance between objects in relative units, calculated based on object size. Let us assume that one such unit corresponds to the length of the measured object (we also assume that for the experiment reasons compared objects should be nearly the same size).

B. *Experimental Results*

During performance evaluation the effect of server load on consistency was evaluated. In the series of experiments consistency metric γ was measured on client processes at various server load. All experiments were run within the 100Mbit/s Ethernet network using computers with 3d-hardware support. The overall performance of computers was sufficient to provide 60 Hz frame rate at test scenario. Test bed included server, two clients to measure ξ metric and additional clients to generate extra traffic. Test scenario assumed virtual environment including two objects moving along predefined trajectories (simulating jet-fighters). Server load was gradually changed by connecting additional clients, each creating new objects and thus producing extra traffic. Fig. 7 shows how consistency (γ metric) depends on server input traffic.

Let us set $\xi = 0.6$ units as a required level of consistency. In this case from the Fig. 7 we can find that maximum allowed server input traffic equals to ~80 Kb/s. This allows us to estimate maximum possible number of components which the system based on the proposed framework can handle simultaneously. For example, if the average traffic produced by one component equals to 1.1 Kb/s the framework can handle up to 70 components (users). Note that this is approximate estimation showing only upper bound of component number, while real number can be lower depending on the particular DVR system features. Also note that during experiments multicasting was disabled, so server was a bottleneck for the entire system. In the near future we plan to estimate scalability of the framework with multicasting enabled.

VIII. WAYS TO IMPROVE SCALABILITY OF THE PROPOSED ARCHITECTURE

A. *Scalability of DVR Systems*

Under the term "scalability" in relation to DVR systems we understand the ability of the system to operate in *a correct way* with the increasing number of users and the extension of the virtual environment (VE). Correct system operation assumes that the system is able *in real time*:

- to maintain data consistency among all the users;
- to process users' interactions;
- to visualize the virtual environment including all users' objects (avatars).

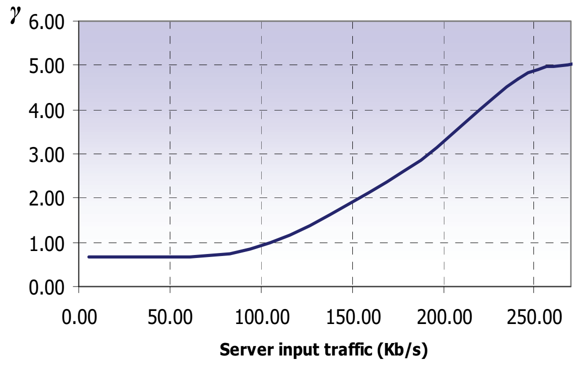


Figure 7. Results of TerraNet performance evaluation

Accordingly, two aspects of DVR systems affecting scalability can be distinguished: the number of users connected to the system (*networking scalability*), the size and complexity of the virtual environment (*computational scalability*).

Increase in the number of users of a DVR system may lead to a lack of network bandwidth due to the considerable increase in network traffic.

Each new user:

- should be sent information about the current VE state on connection;
- adds the additional objects and attributes into the global VE state.

As a result, the number of update messages and the total network utilization are increased. For example, in a peer-to-peer network change in object state at one of N processes will cause $N - 1$ update messages to other processes. If all the processes update their objects states at the same frequency, $O(N^2)$ update messages will be generated in each simulation step.

The computational approach considers the DVR system scalability with increasing complexity of the virtual environment. The complexity of the virtual environment can be measured as the total number of geometric primitives (polygons) that are necessary to render the VE view.

Each new user:

- adds new graphical objects to the virtual environment;
- increases the costs on memory needed to store the states of new objects;
- requires more CPU cycles to process additional state updates and interactions with other users.

Increased complexity of the virtual environment can eventually lead to an overload of graphics subsystem which performance is usually limited to a certain maximum number of polygons that could be rendered per frame. As a result, visualization frame rate is reduced. Frame rates below 30 fps (frames per second) are unacceptable for most interactive virtual reality applications. Some applications, such as modern flight simulators, require an even higher frame rate of 60 fps. The increase in complexity of the virtual environment also requires a better performance of other components of the DVR system nodes, such as CPU and memory.

B. *Discussion*

One of the most commonly used architectures in modern DVR systems is client-server architecture. The main advantages of this architecture are its simplicity and the ability of centralized client monitoring and control. However, the main its drawback is the server which in case of failure or congestion can lead to the entire system failure. For large scale DVR systems a more sophisticated approach should be applied.

Widely used approach to create a more scalable DVR system is the multi-server architecture [16]. This approach is based on the distribution of VE state processing across multiple servers. By adding additional servers it becomes possible to significantly improve overall system scalability and reliability. Each new server allows the system to add new clients and increase the size of the virtual environment. In case of failure of one of the servers, processing of the virtual environment can be redistributed to the remaining servers.

Extended multi-server version of the proposed architecture is shown in Fig. 8. In general, it works as follows. The entire virtual environment is divided into zones of a certain size and shape. Each zone is assigned to a separate server which is responsible for processing corresponding VE partial state and the clients inside it. Connections are established between servers. Each server is connected only with the servers that handle adjacent areas in the virtual environment. Clients are able to migrate from one zone to another. Dedicated servers may be used to manage load balancing and handle connection of new clients.

Even with multi-server architecture implemented there is still one scalability problem related with zone congestion when too many users are crowded in a particular zone. To avoid this, *mirror* servers should be introduced. Each mirror server replicates the VE state inside a zone and processes only some part of its state. In this way, parallelization of zone processing can be achieved.

In order to further improve the reliability and scalability, DVR system architecture can be decentralized by weakening the role of local servers and adding support of direct client process communication [17]. The potential of such *centralized peer-to-peer* system in terms of scalability is especially high. In this respect, latest developments in the field of peer-to-peer file sharing systems, like BitTorrent, Gnutella and Kad [18], may be useful.

IX. Applications and Future Work

The TerraNet framework can be used for various applications requiring interaction of many components in real-time. Several experimental applications were developed using it (see Fig. 9):

- *ShareEdit* – distributed modeling tool prototype, allowing multiple users to build 3d-scenes in real time;
- *FFSystem* – formation flight visualization system [19].

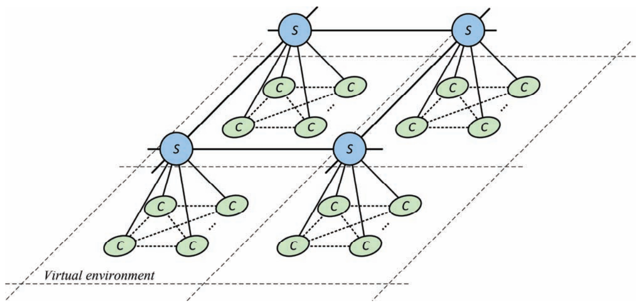
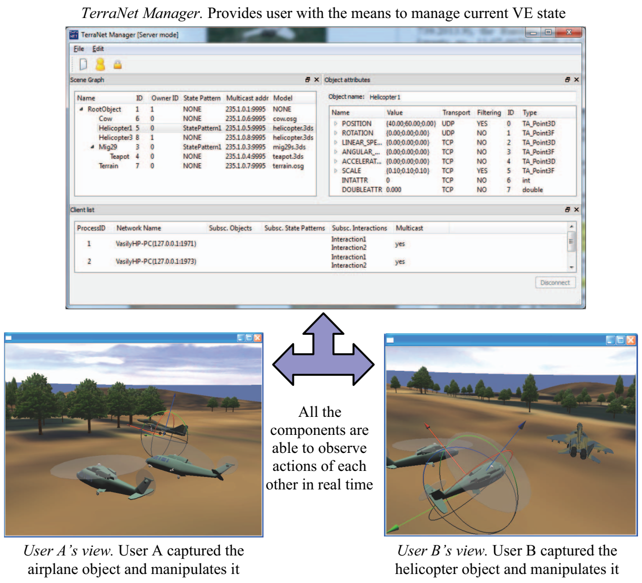


Figure 8. Multi-server architecture prototype. *S* denotes a server process while *C* stands for a client process.

Currently, work is underway on the use of the TerraNet framework as a basis for building contemporary distributed flight simulators, consisting of variety components and providing many pilots cooperative flight (i.e. formation flight). In the near future we are planning to increase the total number of supported clients by adding multi-server


TerraNet Manager. Provides user with the means to manage current VE state
All the components are able to observe actions of each other in real time
User A's view. User A captured the airplane object and manipulates it
User B's view. User B captured the helicopter object and manipulates it
File
Edit
Disconnect

TerraNet Manager [Server mode]

Scene Graph

Name	ID	Owner ID	State Pattern	Multicast addr	Model
RootObject	1	1	NONE	235.1.0.1:9995	NONE
Cow	6	0	NONE	235.1.0.6:9995	cow.osg
Helicopter1	5	0	StatePattern1	235.1.0.5:9995	helicopter.3ds
Helicopter3	8	1	NONE	235.1.0.8:9995	helicopter.3ds
Mig29	3	0	StatePattern1	235.1.0.3:9995	mig29s.3ds
Teapot	4	0	NONE	235.1.0.4:9995	teapot.3ds
Terrain	7	0	NONE	235.1.0.7:9995	terrain.osg

Object attributes

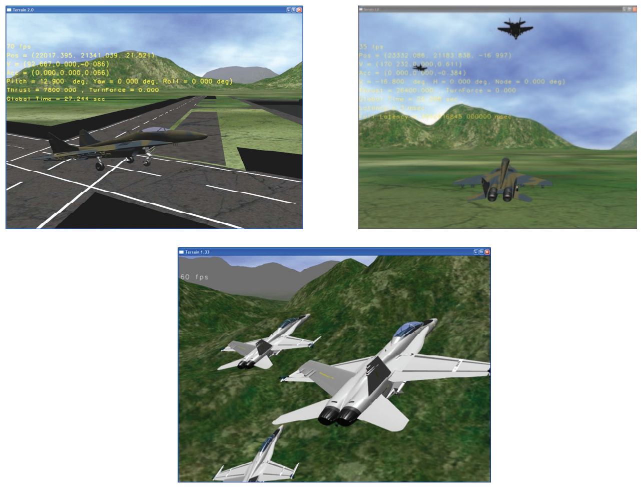
Object name: Helicopter1

Name	Value	Transport	Filtering	ID	Type
POSITION	{40.00;60.00;0.00}	UDP	YES	0	TA_Point3D
ROTATION	{0.00;0.00;0.00}	UDP	NO	1	TA_Point3F
LINEAR_SPE...	{0.00;0.00;0.00}	TCP	NO	2	TA_Point3D
ANGULAR_...	{0.00;0.00;0.00}	TCP	NO	3	TA_Point3F
ACCELERAT...	{0.00;0.00;0.00}	TCP	NO	4	TA_Point3D
SCALE	{0.10;0.10;0.10}	TCP	YES	5	TA_Point3F
INTATTR	0	TCP	NO	6	int
DOUBLEATTR	0.000	TCP	NO	7	double

Client list

ProcessID	Network Name	Subsc. Objects	Subsc. State Patterns	Subsc. Interactions	Multicast
1	VasilyHP-PC(127.0.0.1:1971)			Interaction1 Interaction2	yes
2	VasilyHP-PC(127.0.0.1:1973)			Interaction1 Interaction2	yes

a) ShareEdit – prototype of a distributed modeling tool



b) Formation flight visualization system prototype

Figure 9. Examples of DVR systems created using the TerraNet framework

communication architecture support. Also we are going to investigate the framework performance in more detail. New experiments will cover the influence on system scalability and consistency of such aspects as replication strategies, dead reckoning techniques and clients geographical distribution.

Acknowledgment

This work was partially supported by the Council on Grants of the President of the Russian Federation for State Support of Leading Scientific Schools (SS-316.2012.9) and for State Support of Young Doctors of Science (YD-739.2013.9), the Russian Foundation for Basic Research (grants no. 11-07-00751 and 12-07-00042), and by the Federal Target Program "Research and scientific-pedagogical cadres of innovative Russia" (State contracts 16.740.11.0038 and 16.740.11.0516).

References

[1] S. Singhal, M. Zyda, Networked Virtual Environments: Design and Implementation, ACM Press/Addison-Wesley Publishing Co., New York, NY, 1999.

[2] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules, IEEE Standard 1516-2000 (2000).

[3] M. Capps, D. McGregor, D. Brutzman, and M. Zyda, "NPSNET-V: a new beginning for dynamically extensible virtual environments," in IEEE Computer Graphics & Applications, 20 (5), 2000, pp. 12–15.

[4] C. Greenhalgh, S. Benford, "MASSIVE: a collaborative virtual environment for teleconferencing," in ACM Transactions on Computer-Human Interaction 2, Vol. 3, 1995, pp. 239–261.

[5] H. Tramberend, "Avocado: a distributed virtual reality framework," in Proceedings of the IEEE Virtual Reality Conference, 1999, pp. 14–21.

[6] Second Life Project, URL: http://wiki.secondlife.com.

[7] M. Naef, E. Lamboray, O. Staadt, and M. Gross, "The blue-c distributed scene graph", in Proceedings of the Workshop on Virtual Environments EGVE '03, Vol. 39, ACM, New York, NY, USA, 2003, pp. 125–133.

[8] A. Tanenbaum, M. Steen, Distributed Systems: Principles and Paradigms, 2nd Edition, Prentice Hall, New Jersey, USA, 2006.

[9] D. L. Mills, "Internet time synchronization: the network time protocol," in IEEE Transactions on Communications, 39(10), 1991, pp. 1482–1493.

[10] A. Steed, M. F. Oliveira, Networked Graphics: Building Networked Games and Virtual Environments, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[11] V.Y. Kharitonov, "Motion-aware adaptive dead reckoning algorithm for collaborative virtual environments," in Proceedings of ACM SIGGRAPH VRCAI 2012, Singapore, December 2–4, 2012, ACM, New York, NY, USA, 2012, pp. 255–261.

[12] S. Strassburger, T. Schulze, R. Fujimoto, "Future trends in distributed simulation and distributed virtual environments: results of a peer study", in Proceedings of the 40th Conference on Winter Simulation, Miami, Florida, USA, 2008, pp. 777–785.

[13] OpenSceneGraph – an open source high performance 3D graphics toolkit, http://www.openscenegraph.org.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[15] V.Y. Kharitonov, "A consistency model for distributed virtual reality systems", in Proceedings of 4th International Conference on Dependability of Computer Systems DepCoS-RELCOMEX 2009, IEEE Computer Society, Los Alamitos, CA, USA, 2009, pp. 271–278.

[16] S. Gorlatch, F. Glinka, A. Ploss, and D. Meiländer, "Designing Multiplayer Online Games Using the Real-Time Framework," in Algorithmic and Architectural Gaming Design: Implementation and Development, ed. Ashok Kumar, Jim Etheredge and Aaron Boudreaux, IGI Global, 2012, pp. 290-321, doi:10.4018/978-1-4666-1634-9.ch012.

[17] J. L. Miller, "Distributed virtual environment scalability and security," Technical Report, University of Cambridge Computer Laboratory, October 2011.

[18] S. Androutsellis-Theotokis, and D. Spinellis, "A survey of peer-to-peer content distribution technologies," ACM Comput. Surv. 36, 4, 2004, pp. 335–371.

[19] V.Y. Kharitonov, "A software architecture of distributed virtual reality system for formation flight visualization," in Proceedings of 3rd European Conference for Aero-Space Sciences, Versailles, France, July 6–9th, 2009.