

Prototype of a Scalable Tele-Virtual Environment on the Web Using VRML2.0 / JSDA

Project Sponsors: IBM T.J. Watson Research Center, Yorktown Heights - NY.

Authors: W. Furmanski, D. Dias, B. Natrajan, V. Mehra, S. Pallickara.

Principal Investigator: Dr. Wojtek Furmanski.

**Included as a part of JSDA distribution by Javasoft.

Abstract

The Web is fueling the newest computing revolution and along with the Browser has enabled the dissemination of rich multimedia content to an unprecedented number of users. This has placed new demands on application developers to provide highly scalable applications (tens of thousands of users) to manage new types of information and to deliver it all in good time. Internet Technologies like Java, VRML and the likes, have acted as a catalyst in the development of second generation *scalable* world-wide distributed computing and collaboration environments. A Virtual World by default is not collaborative, viz. If 10 people have loaded the World file from a Web server into their browsers - and they are viewing the file at the same time they wouldn't be aware of each other, this is model akin to the viewing of an HTML file Web page. However if they were to be aware of each others presence and could interact with each other similar to some kind of social activity, we have Tele-Virtual World the denizens of which could take part in a host of activities with each other. These could be playing games or debating the merits/demerits of a certain product in a mall or analyzing complex simulations. In this paper we discuss the design of the Prototype Tele-Virtual World that we have developed. We have used VRML 2.0 and JSDA from JavaSoft to achieve this end.

Introduction

A users' presence in a World scene is usually through a 3D object - traditionally know as an *avatar* - that would interact with other avatars as he deems fit. Any change in an *avatar's* viewpoint is routed to all the participating avatars - people who have loaded the world file in their browsers at the given moment.

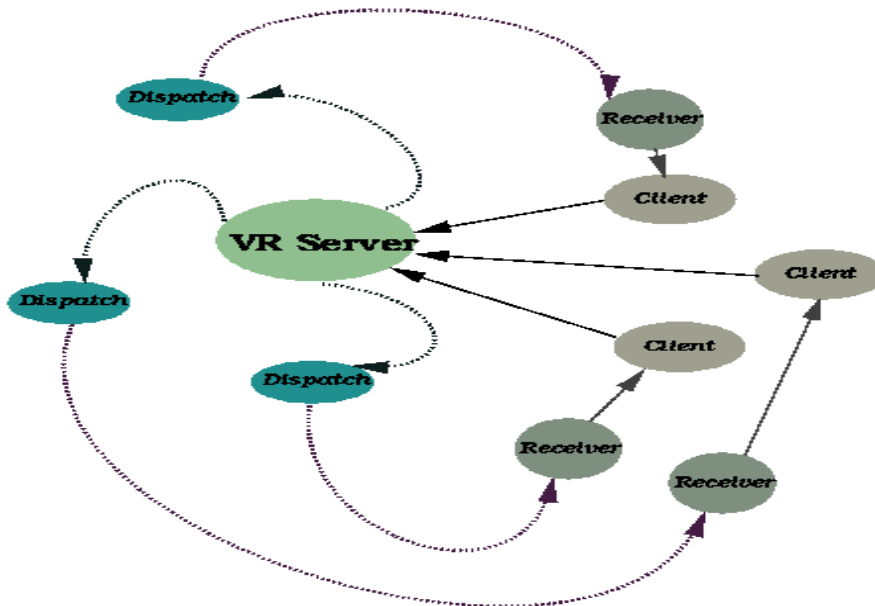
A simple implementation of this would be a Server which waits for connections from clients, and spawns threads (*Dispatch*) to handle any information received from the client regarding updates to position & orientation. It should be understood at this point that the Client in the figure is a Java Script node. Besides this the Server also could assign some sort of identification to each of the clients & also the representation of the avatars.

```

while(true){
    // accept a client's request for connecting to the
Server
    try{
        client_socket = server_socket.accept();
    }catch(IOException e){
        System.out.println("Accept failed: " + " ", " + e);
        System.exit(1);
    }
    System.out.println("Connection established at : " +
client_socket.getInetAddress());

    // create one thread for client request and store it
for interacting with the clients
    id = clients.size();
    clients.insertElementAt(new MuDispatcher(client_socket,
clients, id), id);
    System.out.println(" id=" + id);
}
}

```



Figure[1] - Design of a simple Multi-User VRML World

The clients would in turn spawn threads (*Receivers*) to handle the information received from the thread on the Server side and call the appropriate method in the Client say **updatePositions(...)**, to update the clients would send their positions to the server which would have a thread for each of the clients. Any change in the position of one of the avatars is reflected in all the other VRML-Scenes.

```

try {
    socket = new Socket(HOST, Protocol.PORT);
    in = new DataInputStream(socket.getInputStream());
}

```

```

        out = new
DataOutputStream(socket.getOutputStream());
        id = in.readInt();

        // create a thread to wait for data from server
        new Receiver(in, this);
        connected = true;
    } catch (UnknownHostException e) {
        b.setDescription("Unknown host: " + HOST);
        return;
    } catch (Exception e) {
        b.setDeDescription("Connection error");
        return;
    }
}

```

JSDA - Java Shared Data Architecture

JSDA provides a Shared Framework for Java at the *data level*. Data objects are shared over, specific instances of *Channels*- between two or more *Clients*. To elucidate further Channels are abstractions for data communication paths in JSDA, whereas Clients are objects which are the source or destination of data- in a collaboration environment. Any Client object which needs to register its interest in receiving messages sent over a channel must implement the Channel Consumer Interface. On similar lines if a client is interested in being notified about changes in state of some other object it should implement the Channel Observer Interface. It should be noted that a Client which references a Channel could operate on that channel, instinctively its clear that a Client could operate on Multiple channels by referencing multiple channels at the same time.

Dynamics of Collaboration

To register interest in a certain Channel, a Client first needs to join the Session which the Channel is a part of and then the Channel. A Client could be part of multiple Sessions and thus register interest in Channels across those various Sessions. Ho wever, a client isn't allowed to have more than one consumer on a channel, though replication of this behavior could be done in a roundabout way. As of now the only data that can be shared are byte Arrays, however its possible to share objects too using the Habanero Model or by creating ByteArrayInput/OutputStreams and using the Object serialization from RMI. Its possible to pass complex data type s like images as a sequence of bytes. JSDA also has objects which encapsulate management policies for other given objects. A classic example to this point is the Session Manager authenticating clients to determine if they could join a session.

Functionality supported by the Channel comprises of

- 1) Session Communication : Send message to all Clients in a specific Session.

- 2) Peer : Send message to a specific Client in a certain Session.
- 3) List Client : List all the Clients who have registered interest in a certain Channel.
- 4) List Consumers : List all the Consumers on a certain Channel.

JAVA /VRML2.0.

Decision logic and state management is often needed to decide what effect an event should have on the scene in the VRML world -- " if the password is right, then open the door ". These kind of decisions are expressed as Script Nodes. These nodes receive events from other nodes, process them, and send events to other nodes. A Script node can also keep track of information between execution (i.e. managing internal state over time). In VRML 2.0, Script nodes allow you to fashion nodes that are described by Java classes using the VRML 2.0 Java API. When it comes to programming the actual Java class for this Script node, fields and eventOuts correspond to properties of the class, while eventIns correspond to methods. eventOuts can be referenced by the setValue() method. In addition, there is a constructor method for the class. There is also an eventsProcessed method which will be called after a set of events has been received by the Script node. The essence of Scripting is to add dynamic behaviors to the VRML worlds. The syntax of a Script Node is

```

Script {

    url "Java filename.class "
    EVENTIN eventType eventName
    EVENTOUT eventType eventName
}

```

The url field provides the link between the node and the Java program that will implement the required behavior to be inserted into the VRML scene. VRML 2.0 specifications allow for trapping of events in the scene using various sensors like the TouchSensor, PositionSensor, PlaneSensor. The event on being trapped will be routed as the EVENTIN or the input event to the Script Node by making use of a ROUTE statement.

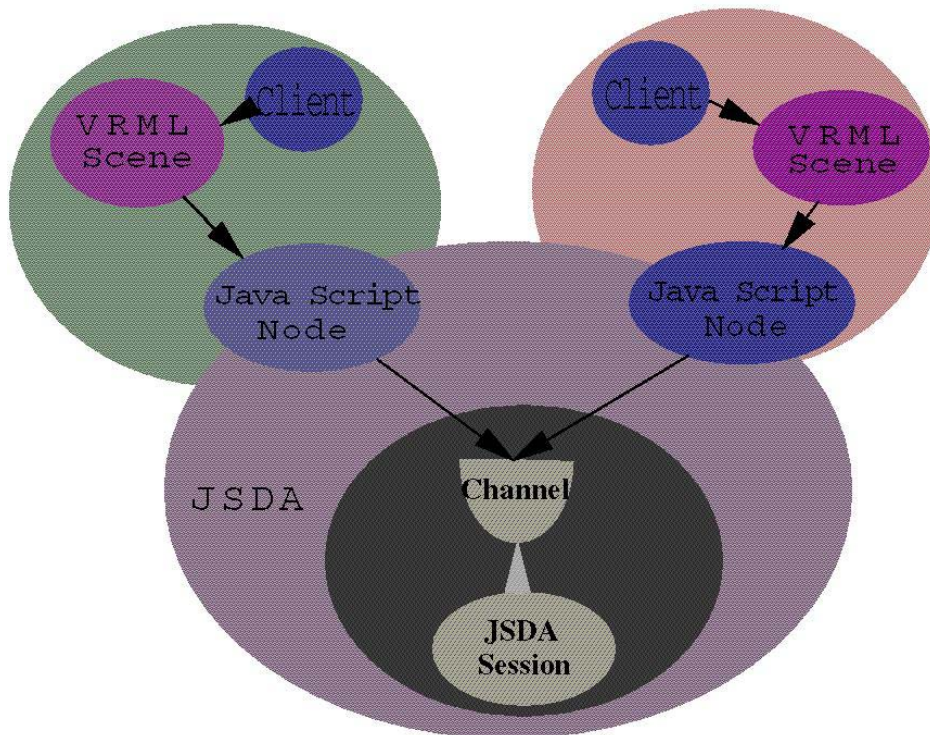
Any events arriving at an EVENTIN field automatically cause the browser to pass the event to the program referred to in the url field of the Script Node. The Java program performs some computation which will add dynamics to the scene and then sends an EVENTOUT or an output event which will then be routed to the required nodes in the scene again through a ROUTE statement.

Script nodes receive events in timestamp order. Any events generated as a result of processing an event are given timestamps corresponding to the event that generated them. Conceptually, it takes no time for a Script node to receive and process an event, even though in practice, it does take some amount of time to execute a Script.

The scripting language (Java) binding may define an initialize method (or constructor). This method is called before any events are generated. Events generated by the initialize method must have timestamps less than any other

events that are generated by the Script node. Similarly, the scripting language (Java) may define a shutdown method (or destructor). This method is called when the corresponding Script node is deleted or the world containing the Script node is exited from. Using Java's powerful networking classes, multi-user VRML worlds can be developed. Using this Script node method of VRML-Java integration, useful applications like Dynamic generation of nodes at runtime can be developed.

JSDA /VRML2.0 - How they interact.



Figure[3]: Interaction of JSDA & VRML Scene. See Figure[4] for details. One starts with writing the VRServer which establishes the Multi-User VRSession and the Channel VRChannel which would be used to route position updates to all the participating clients.

```

try {
    /* Create a session, and publish it. */

    VRSession = new
SharedData.socket.socketSession(sessionName);
    Naming.rebind(url, VRSession);

    /* Get a client-side handle to the session and create a
channel called

```

```

        *"VRChannel". The clients would be using the VRChannel
handles to
        * pass information to various clients.. */

        VRSession = SessionFactory.createSession(url);
        VRSession.createChannel("VRChannel", true);
        System.out.println("Setup and bound VR server.");
    } catch (SharedDataException e) {
        System.out.println("VRServer: main: shared data
exception: " + e);
    }
}

```

The Client - which is the Java Script Node for the World Scene in question would get handles to the VRSession and the VRChannel besides getting references to the *translation* and *rotation* fields of the avatar. This would be done in the **initialize()** method

```

        for (int i=0; i < AVATARS; i++){
            try {
                avatar = (Node)((SFNode)getField("avatar" +
i)).getValue();
                avatarPosition[i] =
(SFVec3f)avatar.getExposedField("translation");
                avatarRotation[i] =
(SFRotation)avatar.getExposedField("rotation");
            } catch (Exception e) {
                b.setDescription("can not get avatar");
                return;
            }
        }
}

```

The VRConsumer receives any data that is sent on the VRChannel, this data is received in the **dataReceived(Data data)** method and calls the performUpdate method in the VRUser.

```

        public synchronized void
        dataReceived(Data data) {
            String message;
            int position = 0;
            int priority = data.getPriority();
            String senderName = data.getSenderName();
            Channel channel = data.getChannel();
            byte[] theData = data.getData();

            message = new String(theData, 0);
            vrUser.commandLine = message;
            vrUser.performUpdate();
        }
}

```

Besides this it would also get handles to the VRSession and the VRChannel and set a Consumer the VRConsumer on that Channel. The connect method would be

```

        String sessionType = "socket";

    try {
        String sessionName = "VRSession";

        /* Create a VR client. */

        client = new VRClient(name);

        /* Resolve the VR session. */

        try {
            String url = "coll://" + hostname + ":" +
hostport +
                "/" + sessionType + "/Session/" +
sessionName;

            session = SessionFactory.createSession(url);

            /* Join the session,channel set the channels data
consumer. */

            session.join(client);
            channel = Channel.join(session, "VRChannel", client);
            VRConsumer = new VRConsumer(client.getName(), this);
            channel.setConsumer(client, VRConsumer);
            } catch (Exception e) {
                return(false);
            }
        } catch (Throwable th) {
            return(true);
        }
    }

```

It is the VRConsumer which is responsible for calling the updatePositions method in the VRUser Java Client. The trick lies in just updating the references to the translation and rotation fields of all the avatars in the World scene based on information sent on the VRChannel by the various Clients.

```

    public void updatePosition(int id, float x, float y, float z){
        System.out.println("Avatar" + id +"on the move ....");
        avatarPosition[id].setValue(x, y, z);
    }

    public void updateOrientation(int id, float x, float y, float
z, float r){
        avatarRotation[id].setValue(x, y, z, r);
    }

    public void performUpdate() {
        StringTokenizer tok;

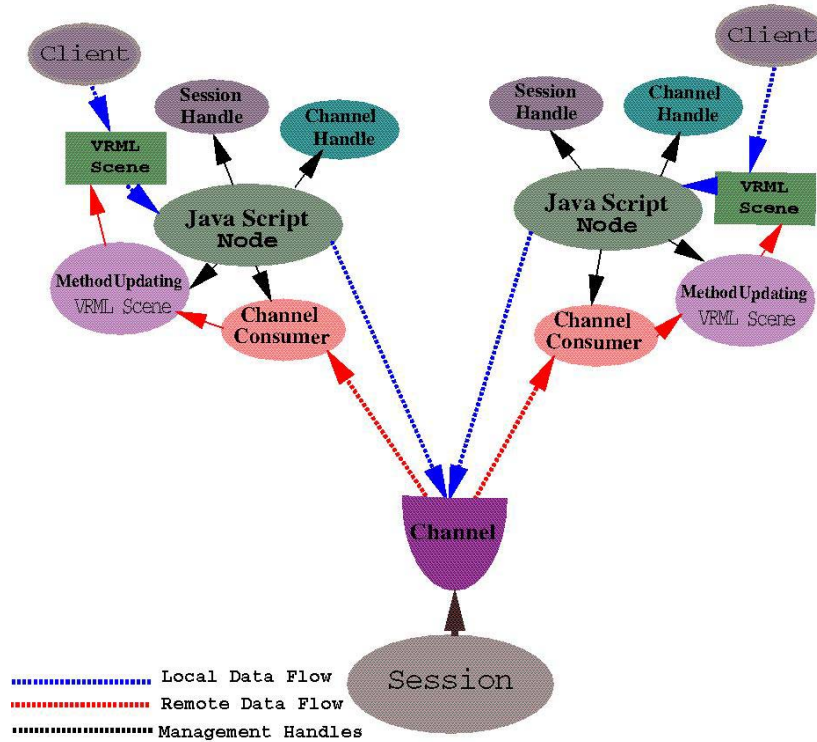
        if (MuProtocol.MOVE == command){
            updatePosition(idOfMove, x, y, z); // update position
        }else{
            r = Float.valueOf(tok.nextToken()).floatValue();
        }
    }

```

```

        updateOrientation(id, x, y, z, r); // update orientation
    }
}
}

```



Collabration Of Virtual Worlds

Figure[4]: *Detailed Interaction of JSDA & VRML Scene*

Conclusions

With the successful culmination of this project and also taking into account the explosive developments in both VRML/Java it should be possible to build massively scalable Tele-Virtual Servers. This project has been a joint effort by the IBM-T.J. Watson Research Center & the NorthEast Parallel Architectures Center at Syracuse University.

References

[1] . [Towards Web/Java based High Performance Distributed Computing - an Evolving Virtual Machine](#) by Geoffrey Fox and Wojtek Furmanski, HPDC-5 Invited Talk, August '96.
 [2]. [Comparison Of Collaboration Environments](#) by Shrideep Pallickara, Vishal Mehra, Wojtek Furmanski, Geoffrey Fox.