An Event-Driven System for Distributed Multimedia Applications

Alésio Pfeifer, Cristina Urrahy, Noemi Rodriguez, and Roberto Ierusalimschy
Computer Science Department – PUC-Rio, Brazil
{alesio,ururahy,noemi,roberto}@inf.puc-rio.br

Abstract

In this work we propose an architecture for distributed multimedia applications based on an event-driven programming model. To avoid the synchronization problems inherent to multi-threaded programming, the proposed architecture is based on a single-threaded structure. Instead of multi-threading, we opted for the event-oriented approach allied to multiple communication channels with user-defined handling procedures to allow the application to deal concurrently with control and data streams. We discuss this programming model, present the system we have implemented based on this model, and describe the experience we have had with this system.

1. Introduction

Over the last years, we have seen a shift in the focus of distributed computing from local-area to large-area networks. However, the programming model that has been largely successful in the context of networks of workstations, which is the client-server model, does not adapt itself well to wide-area distributed systems. In geographically distributed networks, there is no guarantee about the state of the resources, and the synchronous nature of client-server computing becomes inconvenient. This has led to a renewal of the discussion about alternative paradigms for distributed programming [6, 3, 10].

The event-oriented programming paradigm has been gaining importance, due, specially, to the growth in systems with graphical interfaces, which has shifted the control of application flow from the program to the user: Instead of writing active code, that dictates the flow of execution, the programmer writes code to react to user input, letting the user direct the flow of control. More recently, several groups have been investigating the applicability of the event-oriented paradigm to concurrent and distributed programming [7, 8, 4, 2].

ALua is an event-based distributed system that has been successfully employed to build different distributed and parallel applications [12]. In most cases, ALua applications use a dual programming language model, in which a scripting language, Lua [5], is used to coordinate the interaction between components written in C.

We have identified a new class of applications that can benefit from ALua’s programming model: multimedia applications such as video-on-demand, video-conference, or video-telephone. In this kind of application, the transmission of Lua code is useful for transmitting control information, allowing agents to asynchronously receive commands and to exchange code. However, in these applications agents also need to exchange large streams of data. This typically involves synchronous (blocking) operations which do not match well with ALua’s original architecture. An agent that is blocked on an I/O operation may “miss” control events that cancel this same operation, and deadlock may ensue.

In this paper, we propose an event-driven architecture that allows us to deal with both kinds of messages — control and data — in one single asynchronous programming model. We opted for a solution that maintains the single-threaded structure of ALua agents, but at the same time allows agents to handle different communication channels concurrently. This structure fits in well with the requirements of distributed multimedia applications, and at the same time avoids the synchronization problems that are inherent to multi-threaded programming.

2. Multiple communication channels

A program in the ALua system is composed by several processes (also called agents), which may run in different hosts. ALua follows the event-driven design first advanced by GUI systems: Each agent runs an event loop, and only acts as a reaction to some event.

There is only one communication primitive between agents, a send operation, which generates an event in the receiver process. There is no explicit operation to receive messages. The receiver process gets its messages through its intrinsic event loop, like any other event.

What makes ALua quite flexible is what is inside each
message, and how receivers react to communication events. In ALua, each message is a piece of code (written in an interpreted language). When an agent receives a message, it immediately runs its contents. This results in a programming model with the same character of interpreted languages: Not very secure, but highly flexible. The event-driven architecture provides responsiveness, while avoiding internal concurrency and its inherent problems.

In the original ALua architecture, a single callback function that executes Lua code is employed for any message arriving for an agent. Although this has proved to be quite flexible in a range of applications [12], looking at the demands of multimedia distributed applications, we identified the need for an agent to handle different messages in different ways. We thus decided to extend the ALua model and allow it to define different handlers for different messages.

One convenient way to specify that a message is to receive a certain treatment (or service) is through the use of communication channels [1]. In the new ALua architecture, communication channels become objects that the ALua programmer can manipulate. An agent may now define as many communication channels as it needs and associate different callback functions to each of them. The event loop now observes the status of a set of communication channels, and triggers the callback functions associated to the ones that need handling. One channel — called the control channel — is regarded as having a distinguished status, maintaining the original function of exchange of Lua code. We will use the expression Multi-channel ALua to distinguish the new ALua architecture from the original one.

3. Experience with Multi-channel ALua

To experiment with Multi-channel ALua, we have integrated an RTP (Real-time Transport Protocol) library to it [11, 9]. In our ALua-RTP agent, the receive callback function associated to an RTP socket reads the data from the socket and then invokes the appropriate RTP library function. In this way, the RTP library functions, originally designed to be invoked in a synchronous environment, were integrated to the ALua programming model.

To experiment with Multi-channel ALua and RTP, we have set up a sample video application that involves servers, which hand out JPEG streams obtained in real time, clients, which receive such streams and display them on the screen, and proxies, which are clients that are capable of forwarding the received data on to other clients.

We have yet to investigate this application further, tackling situations that demand more of the flexibility it offers and measuring performance. However, this initial experiment showed that the new multi-channel architecture results in an easy-to-use distributed programming tool that fits in well with real-time video applications.

4. Final remarks

In this work we discussed an architecture that allows a single-threaded application to deal concurrently with different streams of information and control. The implementation of the Multi-channel ALua system follows this architecture, associating the flexibility of transmitting Lua code through a main control channel to that of allowing the programmer to define functions for handling data streams on secondary channels.

The ability to configure callbacks adds to the system a flexible and powerful configuration mechanism, in which agents can define functions and associate them with channel callbacks, changing the behavior of the channels to which they are connected, on the fly, according to their needs.

This work was partially supported by CAPES and CNPq.

References