# PROGRAMAÇÃO DISTRIBUÍDA ORIENTADA A EVENTOS

**Aluno: Ricardo Gomes Leal Costa**
**Orientadora: Noemi Rodriguez**

## Introdução

As aplicações geograficamente distribuídas, isto é, que estão espalhadas em diversas máquinas fisicamente distantes e conectadas em rede, precisam tratar de questões como sincronização, tempo de resposta variável e falhas de comunicação, tornando o seu desenvolvimento muito mais complexo e difícil de ser testado.

Com o intuito de facilitar a programação desse tipo de aplicação, desenvolvemos a biblioteca DALua, baseada na linguagem de programação Lua [2] e no sistema ALua [4]. Desta forma, a aplicação é composta por vários processos Lua distribuídos que trocam mensagens pela rede com o uso de recursos providos pelo DALua.

O DALua oferece abstrações de programação para o gerenciamento da entrada e saída de processos na aplicação, envio de mensagens por multicast, tratamento de eventos de erro e facilidades para testes automatizados e depuração.

Para testar a biblioteca, desenvolvemos módulos de suporte a algoritmos distribuídos, provendo exclusão mútua e envio de mensagens com ordenação causal e total. Em seguida, criamos um protótipo de jogo distribuído que utiliza vários desses recursos simultaneamente.

## Objetivos

Nosso objetivo é disponibilizar uma biblioteca para facilitar a programação de aplicações distribuídas. Sua interface deve ser simples e coerente com os algoritmos distribuídos descritos na literatura. Um dos requisitos mais importantes é a tolerância a falhas, devido aos problemas inerentes a sistemas geograficamente distribuídos. A biblioteca deve prover também ferramentas para o desenvolvimento de testes automatizados, garantindo o correto funcionamento da aplicação sob diversas condições de uso.

## Desenvolvimento

O sistema ALua, desenvolvido na PUC-Rio, adiciona suporte à programação distribuída orientada a eventos em Lua. Uma aplicação ALua é composta por processos executando em várias máquinas interligadas em rede. A comunicação entre os processos é feita de maneira assíncrona através da primitiva *send* para envio de mensagens. Não existe uma primitiva de recebimento; a chegada da mensagem é tratada como um evento que implica na execução da mensagem, assumindo-se que esta é um trecho de código Lua.

A biblioteca DALua cria uma abstração sobre o modelo do ALua para gerenciar automaticamente a entrada e saída de processos na aplicação, o envio e recebimento de mensagens com suporte a *multicast* e o tratamento de erros. Dessa forma, torna-se mais fácil aplicar os algoritmos clássicos encontrados na literatura de sistemas distribuídos, evitando os detalhes de baixo nível da implementação. Há ainda uma série de métodos para a criação de testes automatizados, incluindo a simulação de atrasos e erros na troca de mensagens.

Todas as ações efetuadas pelo DALua geram eventos nos processos da aplicação, contendo informações relevantes como, por exemplo, o identificador do processo causador. Os eventos são tratados automaticamente pelo DALua por padrão, mas o usuário tem a opção de definir seu próprio tratador de eventos com o método *dalua.monitor*.

A entrada de processos na aplicação é feita com o método *dalua.enter*. Caso a aplicação ainda não exista, ela é criada nesse momento, caso contrário todos os processos participantes da aplicação são informados da entrada do novo processo. Analogamente, utiliza-se o *dalua.leave* para remover o processo da aplicação.

Para facilitar a criação de um número grande de processos, o método *dalua.spawn* foi adicionado. São especificados o número de processos que se deseja criar na aplicação e o código-fonte a ser carregado em cada processo. Os processos são distribuídos de acordo com o número de máquinas disponíveis na aplicação, tentando balancear ao máximo a quantidade de processos em cada máquina.

O envio de mensagens é efetuado pelo método *dalua.send*. Uma mensagem é composta de um nome de função e os argumentos passados para essa função. Quando a mensagem é recebida, a chamada da função é feita localmente. O destinatário da mensagem pode ser um único processo ou uma lista de processos (multicast) da aplicação.

O tratamento de erros do DALua também funciona pelo sistema de eventos. Cada evento tem um motivo associado, que indica se ocorreu em condições normais ou se foi devido a um erro, como, por exemplo, um *timeout* no envio da mensagem. O tratamento padrão no caso de erro é imprimir na tela a mensagem de erro com sua descrição, mas o usuário pode definir um tratamento especial para tornar sua aplicação tolerante a falhas.

Para facilitar a criação de testes automatizados, desenvolvemos um sistema de temporizadores para agendamento de tarefas. Com ele, é possível agendar chamadas aos métodos do DALua, especificando-se o processo que executará as chamadas, o número de vezes que se deve repetir a chamada e o intervalo de tempo entre cada uma. Este recurso permite simular várias condições de atraso no envio de mensagens para garantir o correto funcionamento da aplicação numa rede real.

**Testes**

Com o intuito de testar o modelo do DALua, criamos três módulos com algoritmos para auxiliar o desenvolvimento de aplicações distribuídas. O de exclusão mútua garante que apenas um processo da aplicação tem acesso a um determinado recurso. Para implementá-lo foi utilizado o algoritmo de Ricart e Agrawala [3], cuja idéia básica é que processos que fizeram pedido para entrar na região crítica efetuam o multicast de uma mensagem de pedido e só conseguem autorização para entrar na região crítica quando todos os outros processos responderem a mensagem. As condições para um processo responder a mensagem são tais que garantem o acesso exclusivo ao recurso.

Os outros dois módulos abordam o problema da entrega ordenada de mensagens [1]. Os protocolos de comunicação que utilizamos, como o TCP/IP, tipicamente só fornecem garantias sobre a ordenação de mensagens com um único destinatário. Algumas aplicações como, por exemplo, bancos de dados replicados, exigem que as mensagens sejam processadas na mesma ordem por todos os elementos de um grupo.

Na ordenação causal, uma mensagem M2 que foi causada em resposta a uma mensagem M1 só é entregue após a mensagem M1. Na ordenação total, a relação causal das mensagens não é importante, desde que todos os processos recebam as mensagens na mesma ordem. Em ambos os casos, é necessário associar *vector timestamps* às mensagens, para identificar a ordem de recebimento.

Para simular um cenário mais realista, optamos por criar um protótipo de jogo MMORPG (*Massive Multiplayer Online Role Playing Game*). Neste tipo de jogo, vários jogadores em máquinas diferentes participam simultaneamente. Cada jogador é representado por um personagem inserido num mundo virtual. Os jogadores podem andar pelo mapa, conversar e se atacar. No caso de um ataque bem sucedido, a vítima é removida do jogo.

Em termos de implementação, cada jogador é um processo da aplicação. Na tela de cada um são exibidos o mapa do jogo e uma linha de comando. Os comandos disponíveis são *join*, para entrar no jogo, *leave*, para sair, *say*, para enviar uma mensagem para os outros jogadores, *move*, para se mover no mapa, e *attack*, para atacar outro jogador. O mapa do jogo é replicado localmente em todos os processos, sendo necessário mantê-lo sempre atualizado.

O jogo exige a utilização de vários recursos oferecidos pelo DALua, incluindo entrada e saída de processos durante a partida, exclusão mútua para gerenciamento de acesso ao mapa do jogo e multicast de mensagens para permitir o bate-papo entre os jogadores.

Para ilustrar o funcionamento do DALua, vamos mostrar o que ocorre quando um jogador utiliza o comando *move n* para se mover na direção norte. A linha de comando chama a função `move`, que recebe como argumento a posição de destino.

```
function move (dest)
    if canMoveTo(dest) then
        mutexProcs = dalua.processes(“game”)
        mutex.enter_cs(mutexProcs, "sendMove", dest)
    end
end

function canMoveTo (pos)
    if pos.x < 1 or pos.x > MAP_SIZE_X
      or pos.y < 1 or pos.y > MAP_SIZE_Y
      or map[pos.y][pos.x].id == MAP_TILE_BLOCKED
      or table.getn(map[pos.y][pos.x].players) > 0 then
        return false
    end
    return true
end
```

A função `canMoveTo` verifica se é possível mover o jogador para a posição desejada. Motivos que impediriam o movimento são a presença de obstáculos ou outros jogadores naquela posição, ou uma posição situada além dos limites do mapa.

O passo seguinte é requisitar acesso exclusivo ao mapa através do módulo de exclusão mútua do DALua (mutex). Isto é necessário porque mais de um jogador poderia tentar se mover para a mesma posição simultaneamente, ocasionando condições de corrida para a modificação do mapa. A variável global `mutexProcs` indica quais processos participarão da exclusão mútua, que no caso são todos os processos da aplicação “game”. Quando o processo que chamou a função `mutex.enter_cs` conseguir o acesso exclusivo, a função `sendMove` será chamada com o argumento `dest`.

```
function sendMove (dest)
    if canMoveTo(dest) then
        dalua.send(mutexProcs, "movePlayer", myID, myPos, dest)
    else
        mutex.leave_cs(mutexProcs)
    end
end
```

Novamente verificamos se o movimento é válido, já que um outro processo pode ter modificado o mapa enquanto aguardávamos o acesso exclusivo. Caso não seja mais possível se movimentar, o acesso ao mapa é liberado para os outros processos com `mutex.leave_cs`.

A função `dalua.send` envia para todos os processos envolvidos, através de multicast, um pedido de chamada de função `movePlayer`, recebendo os argumentos `myID`, que é o identificador único de cada jogador, `myPos`, que é a posição de origem, e `dest`, que é a posição de destino.

```
function movePlayer (player, fromPos, toPos)
    if player == myID then
        myPos = toPos
    end
    removePlayerFromMap(player, fromPos)
    table.insert(map[toPos.y][toPos.x].players, player)
    dalua.send(player, "movePlayerACK")
    redraw()
end
```

A função `movePlayer` atualiza a variável `myPos` caso ele seja o próprio jogador que está se movendo. O personagem é removido da posição de origem no mapa e inserido na posição de destino. Uma confirmação é então enviada de volta em forma de chamada à função `movePlayerACK`. Finalmente, a tela de cada jogador é atualizada com `redraw()`.

```
function movePlayerACK ()
    movePlayerCounter = movePlayerCounter + 1
    if movePlayerCounter == table.getn(mutexProcs) then
        mutex.leave_cs(mutexProcs)
        movePlayerCounter = 0
    end
end
```

Conforme as confirmações de que o mapa foi atualizado são recebidas, o contador `movePlayerCounter` é incrementado até que todos os processos tenham enviado a confirmação. Então, o acesso exclusivo ao mapa é liberado.

O sistema de eventos pode, por exemplo, gerenciar a entrada e saída de jogadores, exibindo uma mensagem informativa na tela de cada jogador e chamando funções responsáveis por requisitar exclusão mútua e atualizar o mapa. Para isso, redefinimos o tratador de eventos para o *join* e o *leave* no início do programa com nossas próprias funções:

```
dalua.monitor(“join”, onJoin)
dalua.monitor(“leave”, onLeave)

function onJoin (event, id, reason)
    print(“* Player “..id..” has joined the game.”)
    addPlayer(id)
end

function onLeave (event, id, reason)
    print(“* Player “..id..” has left the game.”)
    removePlayer(id)
end
```

Os temporizadores podem ser utilizados para simular as ações dos jogadores a fim de criar testes automatizados. O exemplo a seguir cria temporizadores para simular a ação de dois jogadores.

```
function testPlayerAttack ()
    -- jogador 1 se move para (5, 3) uma vez após 1 segundo
    dalua.addtimer(“game”, player1, 1, 1, “move”, {x = 5, y = 3})
    -- jogador 2 se move para (4, 3) uma vez após 2 segundos
    dalua.addtimer(“game”, player2, 2, 1, “move”, {x = 4, y = 3})
    -- jogador 2 ataca o jogador em (5, 3) após 3 segundos
    dalua.addtimer(“game”, player2, 3, 1, “attack”, {x = 5, y = 3})
    -- jogador 2 comemora a morte do jogador 1 após 4 segundos
    dalua.addtimer(“game”, player2, 4, 1, “say”, “ganhei!”)
end
```

## Aplicações

O DALua foi utilizado em 06.1 numa disciplina de pós-graduação do Depto. de Informática da PUC-Rio como ferramenta de aprendizado, permitindo experiência prática com algoritmos distribuídos e arquiteturas peer-2-peer.

## Conclusões

A biblioteca DALua mostrou-se útil no desenvolvimento de aplicações geograficamente distribuídas. Com o modelo de orientação a eventos, o usuário pode facilmente tratar os erros para tornar sua aplicação tolerante a falhas. Os testes automatizados podem simular uma série de problemas na troca de mensagens que são comuns nesse tipo de aplicação.

O desenvolvimento do DALua deu a oportunidade de estudar vários modelos de sistemas distribuídos tolerantes a falhas para a definição do modelo adotado na nossa biblioteca. Permitiu também um melhor entendimento dos algoritmos distribuídos clássicos, que foram implementados e testados para verificar a funcionalidade do DALua.

Finalmente, o desenvolvimento da biblioteca nos permitiu um teste mais pesado da própria funcionalidade do ALua, tendo acarretado na identificação e correção de alguns problemas no sistema.

## Referências

1 - G. COULOURIS, J. DOLLIMORE e T. KINDBERG. **Distributed Systems: Concepts and Design**. Addison-Wesley, 2001.

2 - R. IERUSALIMSCHY, L. FIGUEIREDO e W. CELES. Lua - an extensible extension language. **Software: Practice and Experience**, 26(6):635–652, 1996.

3 - M. RAYNAL. **Distributed Algorithms and Protocols**. Wiley, 1988.

4 - C. URURAHY, N. RODRIGUEZ e R. IERUSALIMSCHY. ALua: Flexibility for parallel programming. **Computer Languages**, 28(2):155–180, 2002.