Pontifícia Universidade Católica do Rio de Janeiro  
INF 2545 – Sistemas Distribuídos  
Professora: Noemi                                  2010.1

Trabalho 3  
Sincronização

**Alunos**  
Danilo Moret  
Thiago Manhente de C. Marques

**Arquitetura**

A rede montada para desenvolver e testar a biblioteca t3lua consiste de uma rede alua com daemons e processos conforme descritos na documentação da biblioteca alua. Na rede t3lua, no entanto, cada daemon da rede recebe funções para gerência de grupos e repasse das mensagens. Sendo assim todos os daemons linkados devem incluir o código existente em t3daemonscode.lua . Isto é realizado no script que monta uma rede inicial em t3daemons.lua , que abre cinco processos lua e inicia um daemon em cada um.

Na arquitetura escolhida são os daemons quem mantém o mapeamento dos processos e grupos existentes, replicando a informação por todos eles. Desta forma sempre que um novo processo se comunica através do t3lua, a biblioteca repassa a comunicação ao daemon do processo e este é responsável por repetir para todos os processos do grupo, ou informar os outros daemons da entrada ou saída do processo em um grupo.

Esta decisão foi tomada para simplificar o trabalho de manutenção das tabelas de grupos. Apesar de existir o problema de replicação da informação por todos os daemons (por exemplo, em uma rede onde a proporção entre daemons e processos seja alta esta estratégia poderia ser repensada), isto evita a centralização desta informação em um único processo ou daemon e a complexidade de localizar onde a tabela de grupos está. Além disso a replicação entre daemons é mais simples, uma vez que a biblioteca alua permite a listagem de daemons da rede, mas não dos processos.

**Desenho da API**

A construção da API foi feita concentrando nos métodos para gerência de grupos. O desenho levou então para a construção dos seguintes métodos:

**t3lua.init(listenFunction[, cbf, debugMode, bogusMode])**

Este método deve ser chamado antes de usar a API. Ele inicia a conexão do processo com um dos daemons conhecidos listados em t3hosts.lua, escolhido aleatoriamente. O método inicia o loop de eventos ao final da conexão, bloqueando o fluxo de execução. Portanto as funcionalidades do programa devem ser desenvolvidas usando o método de escuta passado ou a função de callback opcional. Após ser iniciado o parâmetro t3lua.id fica disponível, sendo uma cópia do valor do identificador único do processo alua correspondente. Os outros métodos irão sair com erro caso esta função não tenha sido chamada antes, ou se a conexão tiver falhado.

Parâmetros:

**listenFunction**, tipo function, obrigatório - função que recebe as comunicações enviadas para este processo. A mensagem é recebida como um único parâmetro no formato de uma tabela com os seguintes campos:

**data**, tipo string - a mensagem transmitida.

**group,** tipo string - nome do grupo onde a mensagem foi transmitida.

**src**, tipo string - id do processo que enviou a mensagem.

**cbf**, tipo function - função de callback executada ao final da conexão e configuração. Não recebe nenhum parâmetro.

**debugMode**, tipo boolean - indica se a biblioteca deve entrar no modo debug, imprimindo mais mensagens que o normal.

**bogusMode**, tipo boolean - indica se a biblioteca deve entrar no modo comparativo, desviando as chamadas “reais” das funções de sincronização por chamadas fictícias, permitindo notar que o comportamento esperado está realmente sendo observado.

**t3lua.join(group[, cbf])**

Adiciona o processo atual ao grupo indicado. Caso o grupo não exista ele será criado.

Parâmetros:

**group**, tipo string, obrigatório - o nome do grupo, que será criado caso não exista na rede.

**cbf**, tipo function - função de callback executada ao final de se unir ao grupo. Não recebe nenhum parâmetro.

**t3lua.initAndJoin(group, listenFunction[, cbf, debugMode, bogusMode])**

Executa em sequencia os métodos de início e adicionar ao grupo atual. Veja os parâmetros nas funções init e join.

**t3lua.leave(group[, cbf])**

Retira o processo atual do grupo indicado.

Parâmetros:

**group**, tipo string, obrigatório - o nome do grupo do qual o processo atual que sair.

**cbf**, tipo function - função de callback executada ao final de se unir ao grupo. Não recebe nenhum parâmetro.

**t3lua.send(group, data[, cbf])**

Método de broadcast de mensagem dentro do grupo. Envia a mensagem sem nenhum controle de sincronização para todos os processos membros do grupo.

Parâmetros:

**group**, tipo string, obrigatório - o nome do grupo onde a mensagem será transmitida.

**data**, tipo string, obrigatório - a mensagem a ser transmitida, em texto puro. Atenção, a mensagem será concatenada ao método send do alua, ela não está protegida, portanto deve se tomar cuidado para ela não fechar a função.

**cbf**, tipo function - função de callback executada ao final de se unir ao grupo. Não recebe nenhum parâmetro.

**t3lua.sendTotal(group, data[, cbf])**

Método de broadcast de mensagem com ordenação total dentro do grupo. Envia a mensagem com controle de sincronização do tipo total, ou seja, garantindo que todos os membros do grupo receberão as mensagens na mesma ordem.

Parâmetros:

**group**, tipo string, obrigatório - o nome do grupo onde a mensagem será transmitida.

**data**, tipo string, obrigatório - a mensagem a ser transmitida, em texto puro. Atenção, a mensagem será concatenada ao método send do alua, ela não está protegida, portanto deve se tomar cuidado para ela não fechar a função.

**cbf**, tipo function - função de callback executada ao final de se unir ao grupo. Não recebe nenhum parâmetro.

**t3lua.id**

Parâmetro preenchido após o processo ter sido conectado ao daemon. Ele recebe o identificador único do processo, sendo uma cópia do id recebido na biblioteca alua.

**Módulo t3lua.mutex**

O módulo de exclusão mútua (mutex) permite o controle de acesso a regiões críticas, através de um mecanismo de semáforos binários.  
Antes de entrar em uma região crítica, o processo deve solicitar acesso à ela, através de solicitações de acesso aos identificadores de exclusão mútua daquela região crítica (pela função **t3lua.mutex.enter()**). Após sair da região crítica, o processo deve liberar os identificadores de exclusão mútua com a função **t3lua.mutex.leave()**.

**t3lua.mutex.enter(group, mutex, cb[, cb\_err])**

Requer acesso a um identificador de exclusão mútua no grupo para entrar em uma região crítica.

Uma solicitação pelo identificador **mutex** será enviada para cada membro do grupo **group**. Quando todos os membros autorizarem, a função **cb** será chamada. Caso algum membro negue o pedido, a função **cb\_err** será chamada, caso fornecida.

As funções **cb** e **cb\_err** não devem possuir argumentos. Caso queira registrar uma função passando argumentos para ela, veja t3lua.mutex.wrap().

Parâmetros:

**group**, tipo string, obrigatório - Nome do grupo em que o acesso ao identificador de exclusão mútua será solicitado.

**mutex**, tipo string, obrigatório - Nome do identificador de exclusão mútua a ser solicitado.

**cb**, tipo function, obrigatório - Função a ser chamada quando o acesso ao identificador de exclusão mútua for garantido.

**cb\_err**, tipo function, opcional - Função a ser chamada caso o acesso ao identificador de exclusão mútua for recusado.

**t3lua.mutex.leave(group, mutex)**

Libera o identificador de exclusão mútua no grupo para acesso por outros processos.

Caso essa função seja chamada no meio de um processo de requisição de acesso (entre a chamada de **t3lua.mutex.enter()** e a chamada de **cb()** ou **cb\_err()**), o processo de requisição de acesso e interrompido.

Parâmetros:

**group**, tipo string, obrigatório - Nome do grupo em que se encontra o identificador de exclusão mútua a ser liberado.

**mutex**, tipo string, obrigatório - Nome do identificador de exclusão mútua a ser liberado.

**t3lua.mutex.wrap(func, ...)**

Encapsula uma chamada de função com argumentos em uma função temporária sem parâmetros.

A função temporária retornada pode ser usada com **t3lua.mutex.enter()**, permitindo assim registrar funções com argumentos para serem chamadas ao receber acesso ao identificador de exclusão mútua.

Parâmetros:

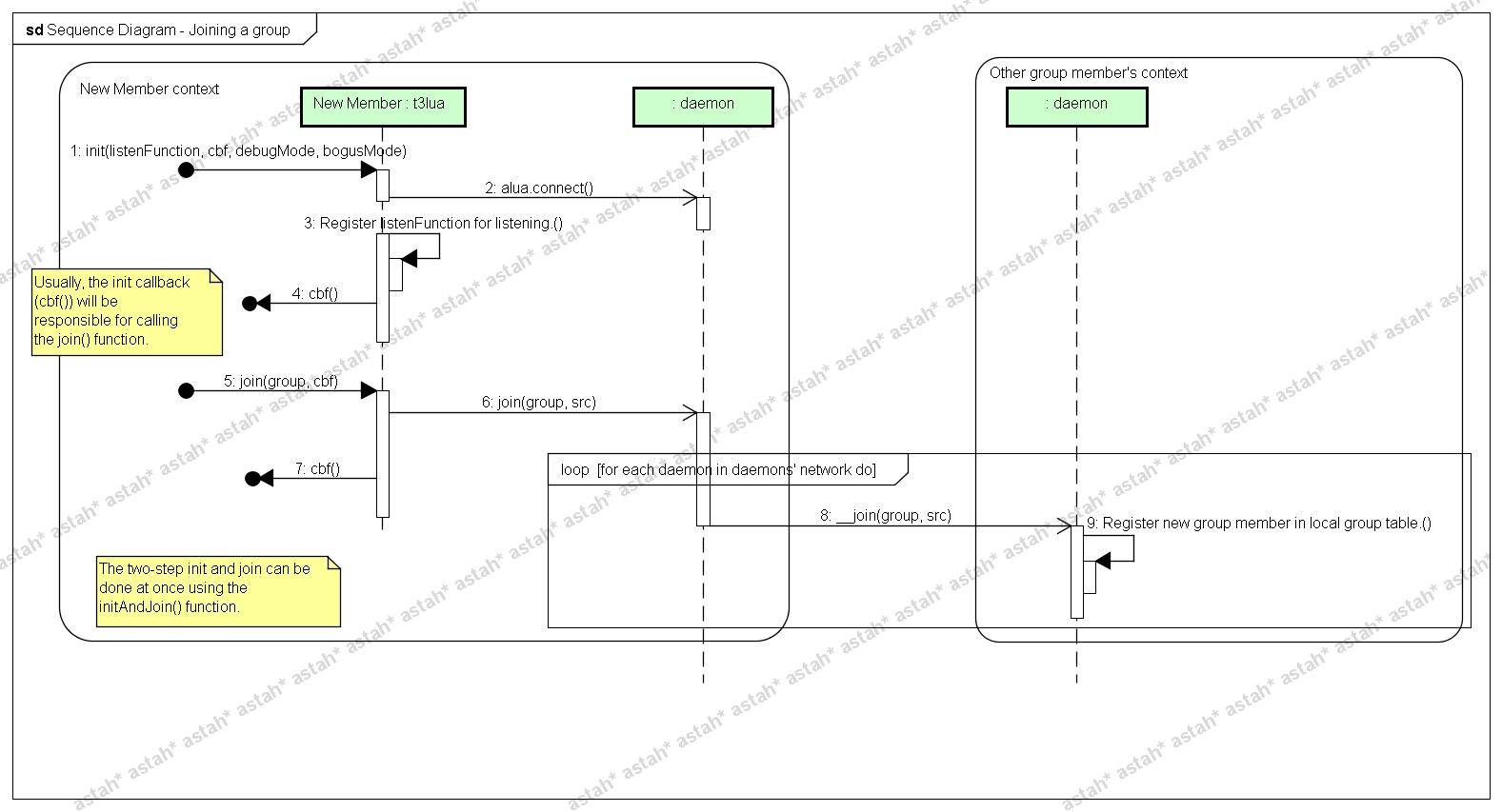
**func**, tipo function, obrigatório - Função a ser chamada.

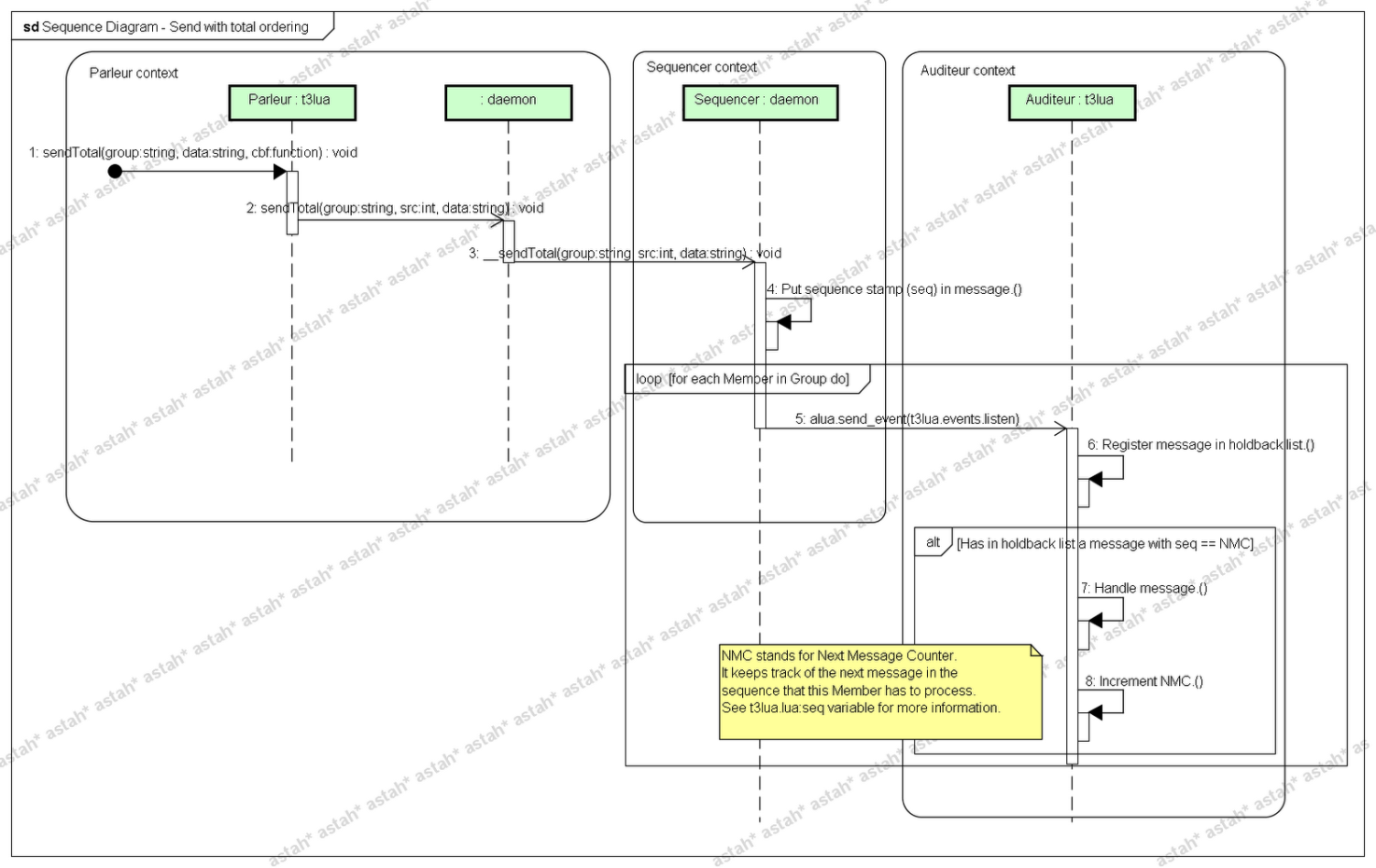
**...** - Argumentos a serem passados ao chamar a função **func**.

**Uso da API**

A seguir temos a descrição de três usos da API, acompanhados de diagramas.

O primeiro cenário é o de iniciar a conexão e entrar em grupo. Ao realizar esta operação o processo cliente está acionando a biblioteca t3lua, que por usa vez irá conectar o processo a um daemon escolhido aleatoriamente na lista de daemons conhecidos. Tanto o processo t3lua quanto o daemons t3lua são por sua vez construídos diretamente sobre um processo e um daemon alua.

Pelas características da própria biblioteca alua, isso quer dizer que ao final da conexão para começar a receber as mensagens e eventos de outros processos é necessário entrar no loop de eventos. O loop de eventos do alua não retorna quando chamado, e as funcionalidades do programa devem ser implementadas no método de callback da conexão ou nos eventos cadastrados. Como o foco é o envio de mensagens, a biblioteca t3lua manteve o callback e usou a API de eventos para cadastrar um evento de transmissão. Sendo assim quando o processo t3lua conecta na rede ele indica um método de escuta que receberá as mensagens quando o evento de recebimento for disparado e as mensagens puderem sair da fila de holdback, abstraindo o conceito de eventos para a camada do processo.

O segundo cenário de exemplo é o do broadcast total. Assumindo que o processo já se uniu ao grupo, para fazer a transmissão garantindo entrega com ordenação total ele chama o método sendTotal da API. Ele indica para este método o grupo e passa a mensagem como string. A biblioteca t3lua então irá repassar a mensagem para o daemon onde o processo está conectado, que por sua vez irá repassa-la para o daemon da rede que foi eleito o sequenciador da rede. O daemon irá numerar a mensagem na ordem que foi recebida e então retransmiti-la para os membros do grupo. Note que o método sendTotal pode receber uma função de callback opcionalmente, mas ela não será chamada somente na recepção da mensagem, e sim logo após que a mensagem for retransmitida para o daemon, permitindo apenas saber na implementação atual se a mensagem foi encaminhada e prosseguir o processamento. Ao invés disso o processo irá ouvir o eco de sua própria mensagem, entregue ordenadamente, através da função de escuta que ele passou como referência ao conectar na rede.

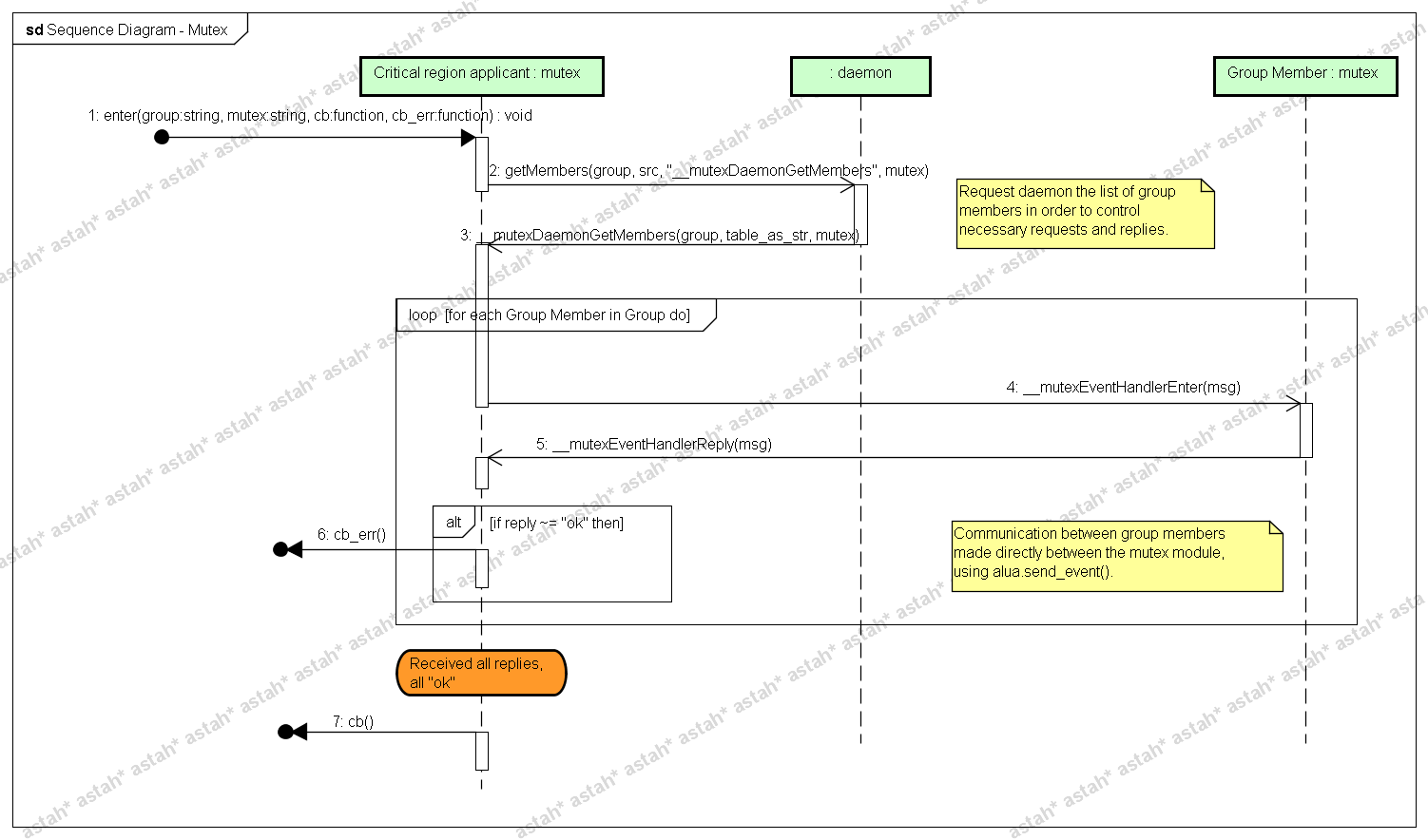
O terceiro cenário ilustra o acesso a uma região crítica através do módulo **t3lua.mutex**

Para acessar uma região crítica, deve-se usar a função **t3lua.mutex.enter()**, passando os nomes do grupo e do identificador de exclusão mútua e as funções a serem chamadas quando o acesso for permitido ou negado.

O módulo de mutex solicitará ao daemon a lista de processos membros do grupo no momento da solicitação de acesso. A lista retornada é armazenada no módulo mutex, de forma que novos membros possam se conectar ao grupo sem afetar as solicitações em andamento.

O módulo mutex envia, então, para cada membro do grupo, o pedido de acesso ao identificador de exclusão mútua. Esse envio é feito através de eventos (uma chamada **alua.send\_event()** passando o evento **t3lua.events.mutex\_enter**). No receptor da requisição, a função **\_\_mutexEventHandlerEnter()**, do próprio módulo mutex, é o responsável por tratar a requisição.

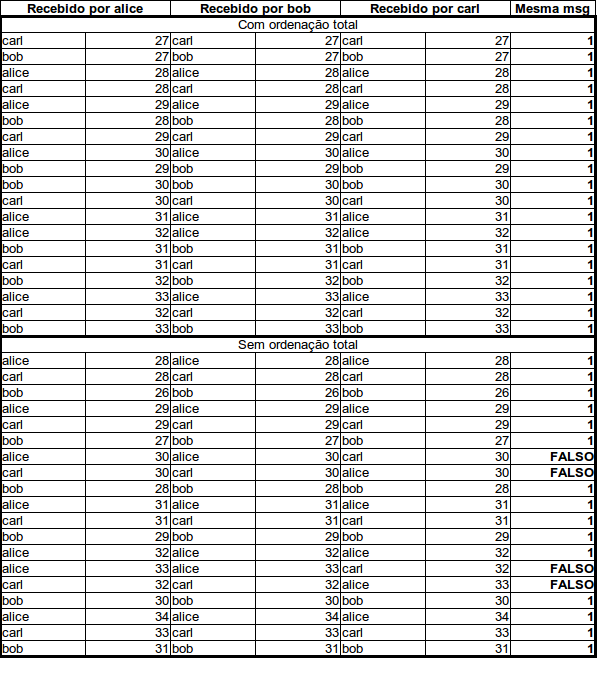
O receptor da solicitação responde também através de eventos, neste caso, enviando o evento **t3lua.events.mutex\_reply**. É enviado “ok”, permitindo ao requerente acesso ao identificador de exclusão mútua, exceto se o próprio processo já tiver acesso ao identificador ou esteja requerendo acesso. Nesses casos, é enviado “no”.

Ao receber todas as respostas esperadas como “ok”, o requerente chama a função **cb()** registrada na ocasião da chamada a **t3lua.mutex.enter()**. Caso alguma resposta “no” seja recebida, o processo de requisição é abortado é chamada a função **cb\_err()** registrada.

**Experimentos e demonstrações**

Como não seria necessário mais simular a performance de máquinas isoladas, apenas confirmar a sequencia de mensagens enviadas ou a exclusão mútua de execução, os experimentos desta vez foram todos realizados na mesma máquina, executando os processos manualmente e em seguida comparando a saída tabulada de cada um. Pela arquitetura escolhida os testes envolvem iniciar um conjunto de daemons usando o script t3daemons.lua e em seguida chamar os scripts que iniciam os processos, entram nos grupos e seguem o roteiro de comunicação para testar cada tipo de comunicação, conforme indicando a seguir.

A principal estratégia de teste foi a sugerida na definição do trabalho. A biblioteca recebe em seu método de início um parâmetro, bogusMode, indicando se a biblioteca deve usar os métodos de efetivos (bogusMode = false, valor padrão) ou desviar para métodos que não fazem os controles (bogusMode = true). O roteiro de transmissão então irá fazer várias comunicações entre os processos, e o resultado quando a biblioteca estiver no modo efetivo deve ser igual ao esperado na definição de cada tipo de sincronização. Já no modo não efetivo deve ser possível encontrar situações onde as condições garantidas pelas sincronizações são violadas.

Para testar o multicast total o script dos processos transmite uma mensagem contendo um nome designando cada processo (alice, bob, carl) e um número serial das mensagens. Ao receber uma mensagem da biblioteca o processo imprime a mensagem para um arquivo próprio (alice.log, bob.log, carl.log) e se a mensagem veio de outro processo o script aguarda entre 1 e 3 segundos (usando uma chamada não bloqueante aproveitando a função socket.select) e então envia outra mensagem para o grupo. A saída dos processos deve ter todas as mensagens recebidas por todos os processos na mesma ordem, mas quando a biblioteca for colocada no modo não efetivo devem existir mensagens recebidas em ordens diferentes.

Quanto ao mutex, apenas um teste simples foi feito rodando várias instâncias do código do arquivo **testmutex.lua** e verificando se eles não entravam na região crítica ao mesmo tempo, tendo resultados positivos. Esses testes apenas verificam se o mutex está funcionando. Um script de teste adicional está sendo confeccionado para mostrar os efeitos na comunicação com o mutex ativado e sem o mutex ativado.

**Limitações e melhorias**

**Instalação do ALua e limitações**

A dificuldade inicial encontrada foi com a instalação da biblioteca alua. A instalação foi realizada com sucesso em uma máquina com o Ubuntu instalado. Um dos pacotes necessários para a instalação do alua é o ossp-uuid, que oferece ao sistema operacional as mesmas funcionalidades da sua biblioteca uuid padrão. Os endereços usados no make da biblioteca luuid, parte da instalação do alua, apontam para o endereço padrão das bibliotecas do sistema, e por isso acabam usando a uuid ao invés da ossp-uuid. Isso foi resolvido editando os arquivos de makefile para apontarem exclusivamente para outro arquivo uuid na hora da compilação. Além do problema da biblioteca, as pastas de libs Lua que vem por padrão configuradas no make da biblioteca apontam para pastas onde o Lua se instalaria caso fosse compilado, e não instalado como pacote no sistema operacional. O arquivo também teve que ser editado para incluir os caminhos das bibliotecas no sistema.

A natureza da biblioteca alua de trabalhar com o modelo de eventos eventos ao invés de multithreading foi também uma dificuldade inicial. A mudança de paradigma não afeta a capacidade de programação uma vez entendida, mas é realmente menos familiar e requer um tempo para se habituar.

Como será visto a seguir nas limitações da implementação, a falta de um timer na biblioteca alua foi sem dúvida a parte mais sentida. Se fosse possível agendar um evento para ser disparado periodicamente o mecânismo de verificação da presença de todos os membros do grupo seria simplificada e também a estratégia de pooling da fila de holdback - apesar desta noção que seria mais simples poder ser do fato da programação seguir outro paradigma. Outra evolução interessante seria a biblioteca oferecer métodos de ping ou mesmo uma verificação periódica de todos os membros da rede. Entendemos que esta última poderia ser inviável em redes muito grandes, e a funcionalidade poderia ser feita no nível da biblioteca de grupos.

**Funcionalidades faltando**

A biblioteca t3lua desenhada não cobre ainda todas as funcionalidades requisitadas. Em especial não foi desenvolvida a comunicação em grupo causal, nem foram realizados testes extensivos usando a capacidade de um processo pertencer a mais de um grupo. A funcionalidade existe, no entanto, e com mais tempo poderia ser melhor explorada. Outros pontos listados a seguir incluem as limitações na fila de mensagens e na verificação que todos os membros da rede estão ativos.

**Limitações e melhorias da implementação**

Sobre a implementação escolhida para o armazenamento das tabelas de grupos e as funções de broadcast propriamente ditas, a escolha foi feita em torno de uma característica da biblioteca alua: os daemons de uma rede conseguem obter a lista de todos os daemons conectados, mas os processos não. Sendo assim foi possível replicar a tabela de grupos por todos os daemons, evitando a criação de um ponto único de falha. Não foi testada a capacidade de aumentar e diminuir a rede, mas a implementação destes controles não parece ser complicada se esta estratégia for seguida - cada novo daemon tem apenas que receber o código dos outros daemons, mais a tabela de grupos ao ser linkada. Outros mecânismos teriam que ser desenvolvidos, como a validação periódica da presença de todos os membros do grupo na rede, e dos próprios daemons. Isto poderia ser feito de forma bem direta usando os eventos da biblioteca alua e um timer.

Além da falta desta funcionalidade, outra melhoria que poderia ser feita é sobre a fila de holdback. A fila de holdback na transmissão em grupo total dentro do t3lua inicialmente estava limitada. Apenas as mensagens do topo da fila eram passadas para o processo. Isso foi alterado e agora a função de escuta deve receber uma coleção de mensagens: a biblioteca irá entregar todas as mensagens válidas ordenadas que já recebeu. Notou-se que isto no entanto não resolveu a questão do processos receberem as mensagens aproximadamente no mesmo tempo: um processo poderia receber as mensagens alice,3 , bob,4 e carl,2 em um momento e outro processo demorar alguns segundos até receber as mesmas mensagens. A ordenação continuava válida. Esta limitação parece vir da execução em várias threads, e cada processo só receberia o evento com a comunicação após os outros na mesma máquina processarem os seus.

Além disso a implementação não cuida do caso de uma mensagem da sequencia ser perdida: caso a mensagem com o próximo id nunca chegue, a fila vai aumentando e nenhuma nova mensagem é entregue. Isto poderia ser resolvido no pooling limitando o tamanho da fila e retornando uma mensagem de erro, ou trocando a estratégia de usar um sequenciador para sincronização descentraliza entre os membros do grupo, aumentando o número de mensagens trafegadas.  
  
**Melhorias no módulo de mutex**  
**Resistência a falhas de membros do grupo**

Embora a implementação lide bem com a entrada de membros no grupo durante a solicitação de acesso aos identificadores de exclusão mútua, o mesmo não é válido para a saída ou falhas de membros dos grupos.

Na implementação atual, caso um membro saia do grupo ou falhe antes de enviar uma resposta para a requisição, o processo requerente entra em bloqueio perpétuo. Seria possível, através de algumas modificações no daemon, permitir que esse informe ao requerente da saída de membros do grupo, de forma que o requerente pudesse remover o antigo membro da lista de respostas a aguardar. Essa alteração evitaria bloqueios no caso de saída de um membro do grupo, mas não foi implementada por falta de tempo.

Já para evitar bloqueios em caso de falha, um temporizador para a requisição seria suficiente, e possivelmente a única solução, visto que alua não possui facilidades de detecção de falhas.

Na implementação atual, em todo caso, um temporizador para as requisições de acesso aos identificadores de exclusão mútua pode ser implementado pelo processo acima do mutex, contornando os problemas de bloqueio.

**Fila de espera por acesso**

Na implementação atual, ao receber uma resposta negativa para uma requisição de acesso a um identificador de exclusão mútua, a função registrada **cb\_err()** é chamada e a requisição é completamente interrompida.

Uma implementação mais rebuscada poderia manter uma fila de espera, de forma que, ao se liberar um identificador de exclusão mútua, o próximo dessa fila seja informado e a ele seja dado o acesso.

Uma forma de contornar o problema na implementação atual é utilizar um temporizador na **cb\_err()** e novamente requerer o acesso até conseguir, ou por um número determio de tentativas.

**Atrelar o identificador de exclusão mútua ao recurso crítico**

A implementação atual oferece controle apenas de acesso aos identificadores de exclusão mútua, mas não do recurso crítico em si. Dessa forma, espera-se um funcionamento correto dos membros do grupo, os que ficam responsáveis em somente acessar o recurso após obter acesso ao identificador de exclusão mútua. Isso torna a solução menos robusta a mau-funcionamentos ou mesmo a membros mal-intencionados.

Uma melhoria no módulo poderia prover uma interface e facilidades para atrelar o acesso ao recurso crítico ao acesso aos identificadores de exclusão mútua, não mais delegando aos membro essa responsabilidade.