Universidade Federal de Santa Catarina – UFSC
Departamento em Informática e Estatística
INE5424-06208B – Sistemas Operacionais II
Semestre 2024/2

Projeto e Implementação de uma Biblioteca para Comunicação Confiável entre Processos

1. Objetivos e Escopo

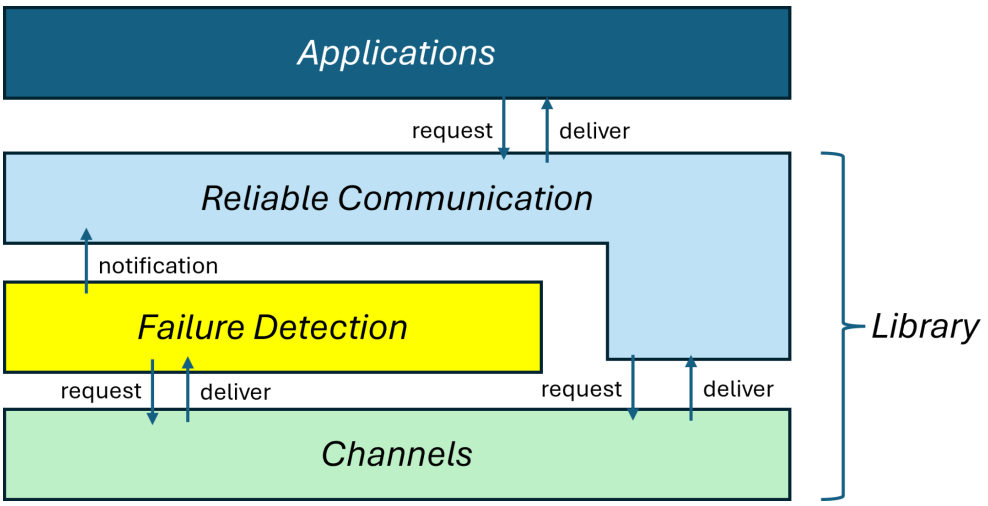
Este projeto consiste em desenvolver uma biblioteca de comunicação capaz de garantir a entrega confiável de mensagens entre os processos participantes de um grupo. Dessa forma, programas que utilizem a biblioteca irão usufruir de garantias na entrega de mensagens, como entrega confiável, difusão com entrega para todos os processos corretos ou nenhum, ou, ainda, garantias de ordem na entrega, como ordenação FIFO e total.

A biblioteca deverá disponibilizar para o usuário as primitivas **`send(id,m)`** e **`receive(m)`** para mensagens destinadas a um processo específico (comunicação 1:1), onde **`id`** é o identificador do destinatário e **`m`** é uma mensagem; e primitivas **`broadcast(m)`** e **`deliver(m)`** para mensagens destinadas a todos os participantes (comunicação 1:n), sendo **`m`** uma mensagem e **`n`** o número total de participantes.

O desafio está em preservar propriedades de entrega confiável em ambientes não confiáveis, onde processos podem falhar e mensagens podem ser perdidas quando transmitidas pelos protocolos de rede subjacentes. A biblioteca deve ser implementada na linguagem C++ e a comunicação entre processos deve ser feita por sockets padrão (*Berkeley sockets / POSIX.1-2008*). Não é admitido o uso de bibliotecas pré-existentes para comunicação confiável ou que apresentem outras abstrações de comunicação para além do uso de sockets padrão.

O conteúdo de mensagens, ilustrado por **`m`** (ex. **`send(id,m)`** e **`broadcast(m)`**), é representado por um *arrray* de bytes, sendo responsabilidade do programador dar contexto ao conteúdo da mensagem. Dessa forma, não há necessidade de implementar mecanismos de serialização/deserialização de tipos de dados que possam ser utilizados nas mensagens.

Com relação à arquitetura de software utilizada no desenvolvimento da biblioteca, será empregada uma estratégia de camadas, favorecendo o reuso. Diferentes camadas da biblioteca são responsáveis por funcionalidades distintas, ex. comunicação confiável, serialização de dados, detecção de falhas, mecanismos de lote (agrupamento de mensagens). A figura a seguir ilustra as camadas da arquitetura.



As aplicações de propósito geral acessam a biblioteca a partir da API disponibilizada pela camada de difusão confiável (*Reliable Communication*), permitindo o envio e recebimento de mensagens com garantias de entrega e ordem. A camada de detecção de defeitos (*Failure Detection*) permite que processos participantes na comunicação monitorem uns aos outros e sinalizem os protocolos de comunicação confiável sobre possíveis saídas de processos do grupo (intencionais ou não-intencionais, no caso de falhas[1]). A camada de comunicação mais baixa, representa os canais de comunicação (*channels*) e implementa *sockets* para comunicação entre os processos participantes.

Para comunicação entre processos utilizando a biblioteca, as funções de comunicação (1:1 e 1:n) devem permitir a troca de mensagens apenas entre os processos de um grupo comunicante. Para este projeto, é assumido um grupo de **`n`** processos, conhecidos antecipadamente. Inicialmente, a biblioteca pode utilizar um arquivo de configuração para designar pares do tipo **`<id, ip:porta>`** representando os processos participantes na comunicação. Posteriormente, o ingresso de processos ao grupo será incorporado a um protocolo de *handshake*. Além disso, a biblioteca deve implementar uma função de inicialização, para que os parâmetros sobre os grupos comunicantes e seus participantes sejam conhecidos.

O quadro a seguir mostra um exemplo de arquivo de configuração, em que 3 nodos participam da comunicação, denominados 0, 1 e 2, com os seus respectivos endereços (ip + porta).

```
nodes = {{0, 127.0.0.1:3000},
         {1, 127.0.0.1:3001},
         {2, 127.0.0.1:3002}};
```

[1] Na literatura, não há um consenso sobre a tradução para os termos *fault*, *error* e *failure*. Nesta descrição assume-se a seguinte terminologia: falha (*fault*), erro (*error*) e defeito (*failure*).

2. Etapas e Critérios de Avaliação do Projeto

A seguir são apresentadas as etapas previstas para o projeto em função das diferentes variações de comunicação que devem ser implementadas neste projeto, bem como os respectivos critérios de avaliação.

2.1. Comunicação 1:1 (funções send e receive)

As mensagens ponto a ponto, do tipo 1:1, devem prover a garantia de *entrega confiável*, ou seja, se um processo **`p`** envia uma mensagem **`m`** para o processo **`q`**, e ambos não falham, então o processo **`q`** recebe a mensagem **`m`** eventualmente[2].

Assuma que os enlaces e protocolos de comunicação subjacentes admitem a perda de um número finito de mensagens e que mensagens espúrias não são criadas pelo meio.

Com isso, as primitivas **`send(id,m)`** e **`receive(m)`** devem implementar o conceito de canal de *comunicação confiável (ou perfeito)*, preservando as propriedades:

- *Entrega confiável*: seja **`p`** um processo que envia uma mensagem **`m`** para um processo **`q`**. Se nem **`p`** nem **`q`** falham, então **`q`** eventualmente entrega **`m`**;
- *Não duplicação*: nenhuma mensagem é entregue por um processo mais do que uma vez;
- *Não criação*: se uma mensagem **`m`** é entregue para um processo **`p`**, então **`m`** foi previamente enviada por algum processo **`q`**.

O protocolo TCP da pilha TCP/IP implementa o conceito de comunicação 1:1 confiável, visto que utiliza mecanismos de ACK para confirmar a entrega de mensagens e números de sequência para evitar a entrega de mensagens duplicadas e fora de ordem. O protocolo UDP da mesma pilha não oferece comunicação confiável, mas pode servir de base para a implementação de protocolos confiáveis. Neste projeto, deve-se utilizar o protocolo UDP para a implementação das primitivas **`send`** e **`receive`**. Dessa forma, será necessário implementar mecanismos de retransmissão e não duplicata na entrega de mensagens.

Há diversos algoritmos na literatura para implementação de canais confiáveis, sendo que os principais se encontram descritos no livro livro Introduction to Reliable Distributed Programming, de Rachidi Guerraoui e Luís Rodrigues[3]. Cada grupo deverá utilizar um algoritmo específico e implementá-lo sobre UDP.

---

[2] Ao longo desta descrição, o termo eventualmente tem a conotação de *eventually*, do inglês, em que um evento garantidamente acontecerá em algum momento futuro. Não confundir com ocasionalmente, em que um evento pode ou não acontecer.

[3] Alguns algoritmos para a implementação de canais confiáveis se encontram sumarizados em https://fileadmin.cs.lth.se/cs/Personal/Amr_Ergawy/dist-algos-slides/fourth-presentation.pdf.

Critérios de avaliação:

- Projeto do mecanismo de comunicação 1:1 com o algoritmo selecionado com, no mínimo, um diagrama de sequência representado a interação entre as respectivas entidades;
- Código fonte C++ das funções integrantes da biblioteca;
- Programa de teste da biblioteca que faça uso das primitivas de comunicação n:1.

2.2. Comunicação 1:n (funções broadcast e deliver)

Para possibilitar difusão de mensagens entre todos os membros do grupo de processos participantes, a API da biblioteca deve disponibilizar as funções **`broadcast(m)`** e **`deliver(m)`**.

Para este projeto, são consideradas 3 formas de difusão confiável:

i. *Best-effort Broadcast (BEB)*: na invocação da primitiva **`broadcast(m)`**, **`m`** é difundida para todos os processos do grupo (incluindo o remetente) e a primitiva **`deliver(m)`** entrega **`m`** se esta não foi entregue ainda.

Mais precisamente, BEB deve satisfazer as seguintes propriedades:

- BEB1 - Validade: se **`p`** e **`q`** são corretos, então cada mensagem difundida por **`p`** é eventualmente entregue por **`q`**;
- BEB2 - Não-duplicação: nenhuma mensagem é entregue mais que uma vez;
- BEB3 - Não criação: nenhuma mensagem é entregue a não ser que tenha sido difundida.

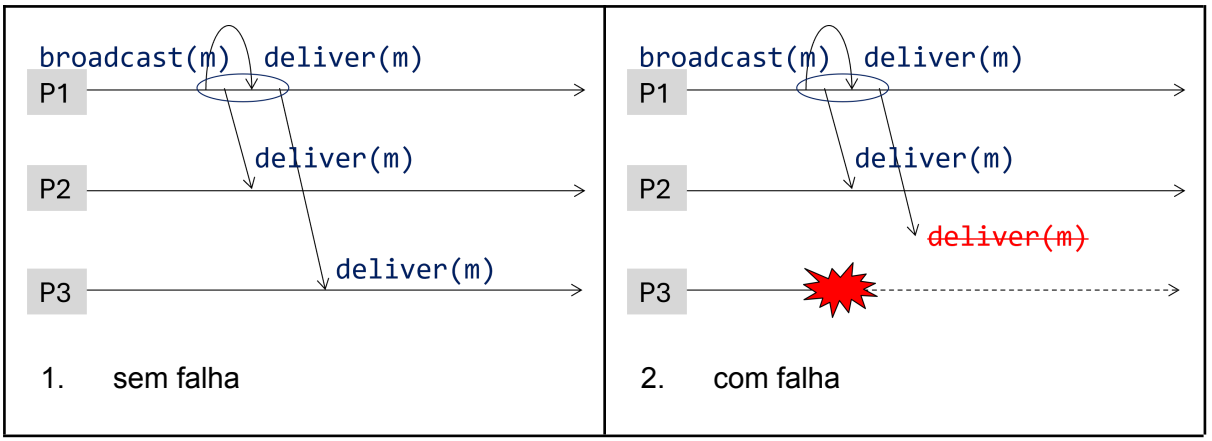
ii. *Uniform Reliable Broadcast (URB)*: Semelhantemente à BEB, na invocação da primitiva **`broadcast(m)`**, **`m`** é difundida para todos os processos do grupo (incluindo o remetente) e a primitiva **`deliver(m)`** entrega **`m`**, mas garante que se algum processo entregou **`m`**, então todos os processos corretos entregarão **`m`**.

Mais precisamente, URB deve satisfazer as seguintes propriedades:

- URB1 = BEB1: (mesma definição de Validade em BEB)
- URB2 = BEB2: (mesma definição de Não-duplicação em BEB)
- URB3 = BEB3: (mesma definição de Não criação em BEB)
- URB4 - Acordo uniforme: para qualquer mensagem **`m`**, se um processo entrega **`m`**, então cada processo correto entrega **`m`**.

Para auxiliar na observação das diferenças na garantia oferecida por BEB e URB, a seguir é apresentada uma figura com traços de execução (a) sem falhas e (b) com a falha de um processo. Para a implementação de difusão usando o BEB, ambos os traços são válidos.

Entretanto, o traço (b) viola a propriedade URB4, logo não é um comportamento válido na implementação do URB.



iii. *Atomic broadcast (AB)*: Esta variação de protocolo de difusão acrescenta ao URB a garantia de ordem total na entrega de mensagens.

Mais precisamente, AB deve satisfazer as seguintes propriedades:

- AB1 = URB1: (mesma definição de Validade em URB)
- AB2 = URB2: (mesma definição de Não-duplicação em URB)
- AB3 = URB3: (mesma definição de Não criação em URB)
- AB4 = URB4: (mesma definição de Acordo uniforme em URB)
- AB5 - Ordem total: se um processo correto entregar **`m1`** antes de **`m2`**, então todos os processos corretos entregarão **`m1`** antes de **`m2`**

A escolha de qual implementação de difusão será executada pode ser feita pelo mesmo arquivo de configuração utilizado na etapa anterior durante a inicialização da biblioteca. O parâmetro broadcast pode ser configurado como **`BEB`**, **`URB`** ou **`AB`**. Este arquivo é o mesmo utilizado para definir os nodos participantes da comunicação, como exemplificado no quadro a seguir.

```
nodes = {{0, 127.0.0.1:3000},
         {1, 127.0.0.1:3001},
         {2, 127.0.0.1:3002}};
broadcast = AB;
```

Note que as soluções previamente implementadas pelo grupo de projetistas desta biblioteca na etapa anterior devem ser reutilizadas nesta etapa: BEB deve ser implementado com as funções **`send(m)`** e **`receive(m)`** já desenvolvidas e URB deve ser desenvolvido utilizando BEB e AB deve utilizar as primitivas do URB, além de **`send(m)`** e **`receive(m)`**, caso necessário. Há diversos algoritmos na literatura para a implementação

de *reliable broadcast* e *atomic broadcast* (baseado em sequenciador, consenso, etc.). O algoritmo utilizado fica à critério do grupo que está projetando a biblioteca. Atenção na implementação do *reliable broadcast*, pois existe a versão com acordo uniforme (exigida neste trabalho) e, simplesmente acordo (não-uniforme). Esta segunda versão é mais permissiva, pois o acordo na entrega de mensagens considera apenas as entregas nos processos corretos.

Critérios de avaliação:

- Projeto do mecanismo de comunicação 1:n com o algoritmo selecionado com, no mínimo, um diagrama de sequência representado a interação entre as respectivas entidades;
- Código fonte C++ das funções integrantes da biblioteca;
- Programa de teste da biblioteca que faça uso das primitivas de comunicação 1:n.

2.3. Injeção de Falhas

A fim de validar o desenvolvimento realizado nas etapas anteriores e também de suportar o desenvolvimento das próximas, os grupos farão ajustes na biblioteca para suportar a injeção de falhas, as quais subjugarão os algoritmos utilizados a situações que evidenciem suas funcionalidades em plenitude.

Dois tipos básicos de falha devem ser modelados e implementados nesta etapa: perda de mensagens e inserção de mensagens corrompidas. Durante a inicialização da biblioteca, dois parâmetros poderão ser informados, um explicitando a porcentagem de mensagens que devem ser descartadas e outro a porcentagem das mensagens que devem ser corrompidas. Cada um dos processos integrantes do grupo atuará também como injetor de falhas, uma vez que tais funcionalidades estão implementadas na camada mais baixa da biblioteca e que, portanto, quando ativadas, atuam transparentemente. Falhas serão configuradas conforme ilustrado a seguir:

```
nodes = {{0, 127.0.0.1:3000},
         {1, 127.0.0.1:3001},
         {2, 127.0.0.1:3002}};
broadcast = AB;
faults = {{drop = 1}, {corrupt = 1}}; // %
```

Critérios de avaliação:

- Logs dos programas de teste da biblioteca desenvolvidos anteriormente demonstrando as falhas injetadas e o comportamento esperado dos algoritmos.

2.4. Detecção de Defeitos

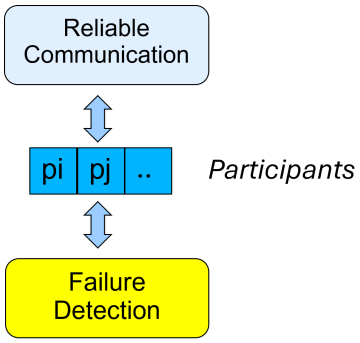
Ao iniciar a biblioteca, um serviço de detecção de defeitos (*failure detection*) deve ser inicializado. Este serviço executa em cada nodo participante na comunicação e

constantemente consulta o estado vital dos demais participantes. A consulta sobre a saúde dos processos é feita por meio de mensagens de *heartbeats*. Cada participante, periodicamente informa aos demais que está *vivo*. Cada processo mantém uma lista de participantes (com base nas configurações de inicialização da biblioteca) e o estado dos participantes, que pode ser: não inicializado, ativo, suspeito, defeituoso.

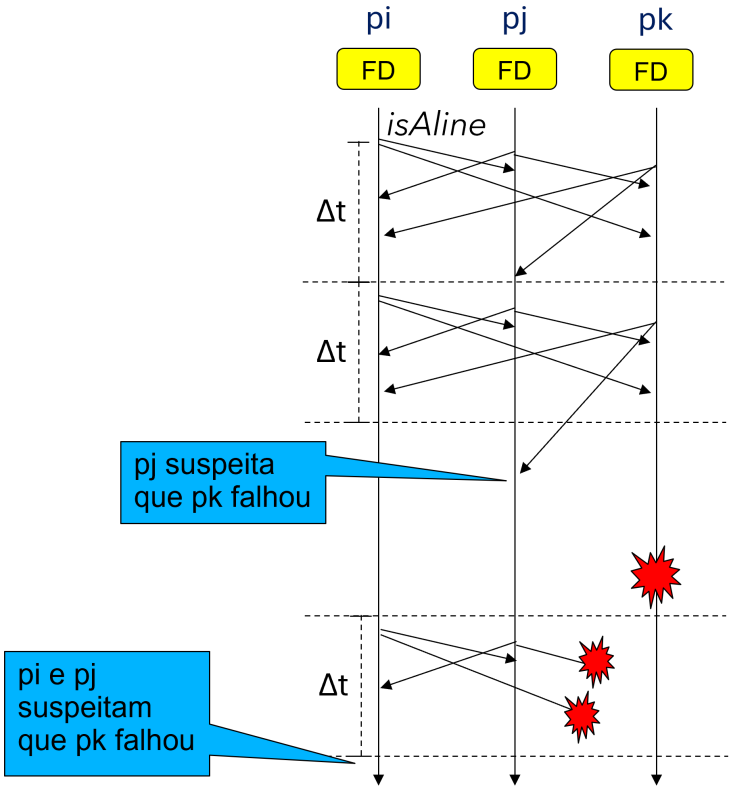
Ao iniciar, cada processo executa um protocolo de *handshaking*, tentando estabelecer contato com os demais participantes do grupo. Caso o processo seja o primeiro a inicializar, não encontrará os demais participantes, pois ainda não foram inicializados. Neste caso, o detector mantém o estado dos demais processos como "*não inicializados*". Um processo em execução, quando recebe o contato de novos participantes, responde que está vivo, fazendo com que os novos processos indiquem este como "*ativo*". Durante o ciclo de vida do processo, este envia mensagens "*estou vivo*", repassando informação sobre suspeitas que possa ter sobre processos defeituosos no sistema (se tiver alguma suspeita). O intervalo entre envios de mensagem "estou vivo" é um parâmetro do sistema e deve ser configurado no arquivo de configuração, em milissegundos, como o exemplo a seguir, com o parâmetro **`alive`** configurado a cada 1s.

```
nodes = {{0, 127.0.0.1:3000},
         {1, 127.0.0.1:3001},
         {2, 127.0.0.1:3002}};
broadcast = AB;
alive = 1000; // ms
faults = {{drop = 1}, {corrupt = 1}}; // %
```

A figura a seguir ilustra (a) a troca de informações sobre o conhecimento dos membros ativos no sistema localmente e (b) a detecção de um processo defeituoso. Tanto a camada de detecção de falhas quanto a de comunicação confiável utilizam uma lista de processos participantes, conforme definição inicial. As propriedades de comunicação confiável devem ser preservadas para todos os processos não defeituosos (garantias de entrega e ordem). Portanto, caso um subconjunto de processos ativos troque mensagens antes que todos os participantes iniciem, estas mensagens deverão ser entregues posteriormente aos processos que atualizem os seus estados de "*não inicializado*" para "*ativo*".



1. Lista de participantes tem estados atualizados pelo detector de falhas e este notifica o serviço de comunicação confiável conforme processos tornem-se ativos, suspeitos ou defeituosos.



2. Diagrama de troca de mensagens com suspeita de defeito em pk por pj, seguida da detecção, quando a maioria dos processos suspeita que pk é defeituoso.

A deteção de defeitos ocorre quando uma maioria de processos suspeita de algum processo (possivelmente defeituoso). Portanto, a configuração do número de participantes deve prever um número de participantes que admita a existência de um quórum de processos de maioria. Por exemplo, assumindo que até **f** processos possam falhar, é necessário ter $n = 2f + 1$ processos no sistema. Por exemplo, se $f = 1$ (significa que no máximo um processo pode falhar), serão necessários $n = 3$ processos. Para tolerar $f = 2$, são necessários $n = 5$ participantes ao todo, e assim sucessivamente.

É necessário atenção nos períodos de inicialização do sistema, pois pode ser que um subconjunto de processos inicie a execução e troque mensagens antes que **n** os processos do grupo tenham iniciado. Neste caso, cada mensagem entregue aos processos ativos deverá ser entregue aos processos que tornem-se ativos após a troca de mensagens, respeitando as propriedades do protocolo de difusão utilizado. Um buffer com mensagens pendentes para os processos "*não inicializados*" deve ser mantido até que os **n** processos tenham atualizado o seu estado para "*ativo*".

Critérios de avaliação:

- Projeto do mecanismo de detecção de defeitos;
- Código fonte C++ das funções adicionadas à biblioteca;
- Programa de teste da biblioteca que evidencie as funcionalidades do detector de defeitos;
- Logs do programa de teste (ou do sistema) demonstrando as funcionalidades do detector de defeitos.

2.5. Grupos Dinâmicos e Falhas Transientes

A biblioteca projetada e implementada nas etapas anteriores assume a utilização por um único grupo fixo, definido no arquivo de configuração. O serviço de detecção de defeitos também assumia que as falhas que levaram ao defeito eram permanentes e que, portanto, um nodo marcado como defeituoso jamais voltaria a integrar o grupo.

Nesta etapa, a biblioteca deve ser expandida para suportar a criação dinâmica de grupos e também o ingresso de nodos aos grupos em tempo de execução. Para tal, o protocolo de *handshake* deve ser modificado para suportar o ingresso em grupos específicos. A criação de grupos pode se dar tanto pela solicitação, por parte de um nodo, de ingresso em um grupo ainda não existente ou quanto por mensagens específicas. A mensagem de ingresso por ser unificada com a mensagem de *heartbeat*, de forma que falhas transientes não impeçam o retorno do nodo ao grupo. O arquivo de configuração inicial pode continuar existindo, principalmente em função dos outros parâmetros, mas também para definir um grupo inicial.

Critérios de avaliação:

- Projeto do mecanismo de criação de grupos e de ingresso de nodos;
- Código fonte C++ das funções adicionadas à biblioteca;
- Programa de teste da biblioteca que evidencie as funcionalidades adicionadas durante esta etapa;
- Logs do programa de teste (ou do sistema) demonstrando as funcionalidades do detector de defeitos.

2.6. Avaliação de Desempenho

Para validar a biblioteca e avaliar seu desempenho, deve ser implementado um serviço do tipo *key-value store* replicado. A ideia é que um conjunto de réplicas mantenha localmente uma tabela *hash* e disponibilize operações do tipo **`write(k,v)`** e **`read(k)`**. A replicação das operações de clientes submetidas ao serviço deve ser feita pelo uso de difusão. Dessa forma, devem ser exploradas as implementações de broadcast (BEB, URB e AB).

Esta aplicação deve ser instrumentada de forma que a vazão do serviço possa ser mensurada a cada segundo, ou seja, deve ser coletado o número de operações realizadas por segundo, desde que o serviço foi inicializado. Para estimular carga no sistema, é

necessário implementar um gerador de carga capaz de produzir requisições do tipo **`write(k,v)`** e **`read(k)`**.

Para efeitos de avaliação de desempenho, o gerador de carga deve permitir as seguintes configurações:

- Percentual de leitura e escrita: indica o percentual de operações **`write(k,v)`** (escrita) e **`read(k)`** (leitura) geradas;
- Número total de operações: indica quantas operações o gerador de carga vai produzir durante a execução;
- Número de clientes: o gerador de carga deve ser uma implementação com múltiplas *threads* trabalhadoras, sendo que cada thread envia uma requisição, dorme por uma fração de segundos (ex. 100ms) e repete este comportamento em um laço de repetição, até que o número total de operações seja alcançado. Quanto mais *threads* simultâneas, maior a carga submetida ao sistema.

A medição de desempenho será efetuada mediante o acréscimo de carga em sucessivas execuções de teste. Pode-se iniciar os testes com poucos clientes, seguido de testes com acréscimos de carga. A observação dos limites de desempenho são constatadas com a percepção de redução no crescimento da vazão com o acréscimo de carga.

Critérios de avaliação:

- Programa de teste de validação da biblioteca com as funções desenvolvidas nesta etapa;
- Logs do programa de teste demonstrando as funcionalidades desenvolvidas nesta etapa;
- Relatório simples de desempenho da biblioteca variando-se os parâmetros que definem as proporções de leitura e escrita, tipo de difusão utilizada, bem como as falhas (token faults no arquivo de configurações).

3. Referências

- Introduction to Reliable Distributed Programming. Rachidi Guerraoui, Luís Rodrigues.
- Distributed Computing Fundamentals, simulations, and Advanced Topics. Hagit Attiya, Jennifer Welch.
- Veríssimo, P.; Rodrigues, L. Distributed Systems for System Architects (1st ed), 2001 (Acessível online: https://pergamum.ufsc.br/acervo/6047213)
- Distributed Systems (3rd edition). Maarten van Steen, et al. (https://www.distributed-systems.net/index.php/books/ds3/)
- Computer Networks: A Systems Approach. Larry Peterson and Bruce Davie (https://book.systemsapproach.org/)
- Slides de aula: Broadcast Algorithms. Björn A. Johnsson (Lund University) https://fileadmin.cs.lth.se/cs/Personal/Amr_Ergawy/dist-algos-slides/fourth-presentation.pdf