**NetWorked Distributed System Simulator**

# 系统功能

本系统用于模拟**NetWorked Distributed System（NDS）**

## 主要功能

1. 根据给定个数的Processors 建立环装连接网络结构，每个处理器仅与相邻的处理器交换信息。
2. 实现了leader election:
   1. 以LCR 算法实现leader election
   2. 以HS 算方法实现leader election
3. 分别对两个leader election 算法进行correctness 和performance评估并对比，其中:
   1. correctness评估主要评估两个算法最终选举leader的正确率
   2. performance 评估包括了time, messages以及rounds三个数据

## Java实现

### NDS主要实现类

NDS主要实现类为 com.courseworks.nds包下的NDSSimulator 类，该类实现了如下功能:

1. 根据给定个数Processors 初始化环装连接网络结构, NDSSimulator类中的initializeNetStructure(num\_processors, idGenerationStrategy, electionStrategy)方法实现了该功能. 参数说明如下:
   1. num\_processors: int 类型, 用于指定网络中processor个数
   2. idGenerationStrategy: NDSSimulator类中定义的枚举类型,用于指定id生成策略, 包括(random, ascend, descend)
   3. electionStrategy: NDSSimulator类中定义的枚举类型,用于指定leader election方式,包括(LCR, HS)
2. 根据给定id生成策略生成id序列(赋给每个processor), NDSSimulator类中的generateRandomNumberSequence(n, k), generateDescendNumberSequence(n, k), generateAscendNumberSequence(n, k)方法实现了该功能, 参数说明如下:
   1. n: int类型，用于指定序列的最大id
   2. k: int类型, 用于指定序列的长度
3. 根据给定leader election 策略实现leader election, NDSSimulatorl类中的getLeader()方法实现了该功能，LCR HS实现参照给定的伪代码

### NDS评估类

NDS评估类位于com.coursework.evaluation 包，主要包括如下功能:

1. correctness评估

该功能由CorrectnessEvaluation类实现,该类的初始化参数为int类型,用于指定最大的迭代次数

其中:

EvaluateLCR方法用于评估LCR算法准确率, 最终返回一个长度为三的数组, 第一个元素表示LCR算法在random策略下的准确率, 第二个元素表示LCR算法在ascend策略下的准确率,第三个元素表示LCR算法在descend策略下的准确率

EvaluateHS方法用于评估HS算法准确率,返回同上。

1. performance评估

该功能由PerformanceEvaluation类实现

其中evaluate()方法用于对LCR算法和HS算法的Performance评估,该方法会分别针对LCR算法和HS算法在RANDOM, ASCEND,DESCEND生成策略的下的Performance评估:

一共n次batch，其中processors个数每个batch 递增500，对于每次batch，又会生成100次不同的id序列, 最终计算每个batch 下LCR算法和HS算法的最小用时，最大用时，平均用时，最小messages,最大messages，平均messages以及平均rounds.

最后，将评估数据写入到指定的excel 文件中

# 评估数据

## Correctness 评估

在对LCR算法和HS算法在RANDOM、ASCEND和DESCEND三种策略下分别进行了100次，500次，1000次，1500次，3000次和5000次评估后，correctness结果如表1所示。

**表1 LCR和HS算法的correctness表**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **LCR** | | | | | | |
| **ID/Processors** | **100** | **500** | **1000** | **1500** | **3000** | **5000** |
| **RANDOM** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **ASCEND** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **DESCEND** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **HS** | | | | | | |
| **ID/Processors** | **100** | **500** | **1000** | **1500** | **3000** | **5000** |
| **RANDOM** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **ASCEND** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |
| **DESCEND** | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 1.0 |

## Performance 评估

### Time 评估

Time 评估主要评估LCR算法和HS算法选择leader所需要的时间。对LCR和HS算法进行了20批次评估，在初始3000个Processors的基础上每批次递增500个processors，每个批次又会产生100次不同的ID序列，最后计算每个批次的最小用时，最大用时以及平均用时。

最终，RANDOM 策略下的LCR算法和HS算法选择leader所需的平均用时结果如图1所示。

**图1 Random策略下的平均用时**

在Processor数量相同的情况下，HS算法选出leader的平均时间要比LCR算法短很多。

ASCEND策略下LCR算法和HS算法选择leader所需的平均用时结果如图2所示。

**图2 Ascend策略下的平均用时**

可以看出，在Processor数量相同的情况下，HS算法选出leader的平均时间要比LCR算法短很多。

DESCEND策略下LCR算法和HS算法选择leader所需的平均用时结果如图3所示。

**图3 Descend策略下的平均用时**

可以看出，在Processor数量相同的情况下，HS算法选出leader的平均时间依旧要比LCR算法短很多。

### Messages 评估

Messages评估主要评估LCR算法和HS算法选择leader一共所需要传送的Messages数量,在Time 评估的同时也会得到message传送数量，记下每个批次的最小传输Messages，最大传输Messages以及平均传输Messages。

最终，RANDOM 策略下的LCR算法和HS算法选择leader所需的平均传输Messages结果如图4所示。

图4 Random策略下的平均传输Messages

可以看出,在Processor相同的情况下，HS算法选出leader所需传输的Messages数量远远少于LCR算法。

ASCEND 策略下的LCR算法和HS算法选择leader所需的平均用时结果如图5所示。

图5 Ascend策略下的平均传输Messages

可以看出,在Processor相同的情况下，HS算法选出leader所需传输的Messages数量远远少于LCR算法。

DESCEND 策略下的LCR算法和HS算法选择leader所需的平均用时结果如图6所示。

图6 Descend策略下的平均传输Messages

可以看出,在Processor相同的情况下，HS算法选出leader所需传输的Messages数量远远少于LCR算法。

### Rounds

使用HS算法选出leader所需要的最小Rounds, 最大Rounds以及平均Rounds如图7所示。

图7 Random策略下HS Rounds分布

使用LCR算法选出leader所需要的最小Rounds, 最大Rounds以及平均Rounds如图8所示。

图8 Random策略下LCR算法 Rounds分布