CSE5020 Assignment 2

1 问题描述

Chandy-Lamport 算法是一个用于记录分布式系统全局一致状态的快照算法。在异步分布式系统中，由于缺乏全局时钟，直接获取系统的全局状态是不可能的。该算法通过 marker 消息的传播，实现了在不停止系统运行的情况下捕获一个一致的全局快照。

本次实验基于普林斯顿大学 COS418 课程的作业模板，使用 Go 语言实现了完整的 Chandy-Lamport 分布式快照协议。

2 算法原理

Chandy-Lamport 算法的核心思想如下：

1. **初始化快照**：当一个进程开始快照时，它记录自己的本地状态，并向所有出站通道发送 marker 消息。
2. **首次接收 marker**：当进程第一次收到 marker 时，它立即记录自己的本地状态，将接收到 marker 的通道状态记录为空，并向所有出站通道发送 marker 消息。
3. **后续接收 marker**：当进程已经记录过状态后再次收到 marker，它记录该通道从记录状态到收到 marker 之间的所有消息。
4. **完成条件**：当进程从所有入站通道都接收到 marker 后，该进程的快照完成。

算法保证了捕获的全局状态是一致的，即不存在”孤儿消息”（在快照中出现但发送者未记录发送的消息）。

3 实现细节

3.1 数据结构设计

为了实现 Chandy-Lamport 算法，在 `Server` 结构体中添加了以下字段：

```go
type Server struct {
    Id             string
    Tokens         int
    sim            *Simulator
    outboundLinks map[string]*Link
    inboundLinks  map[string]*Link
    // 新增字段
    snapshots       *SyncMap  // snapshotId -> *SnapshotState
    markersReceived *SyncMap  // snapshotId -> map[src]bool
}
```

在 Simulator 结构体中添加了:

```go
type Simulator struct {
    time              int
    nextSnapshotId    int
    servers           map[string]*Server
    logger            *Logger
    // 新增字段
    snapshotComplete *SyncMap  // snapshotId -> map[serverId]bool
    snapshotDone     *SyncMap  // snapshotId -> chan bool
}
```

使用 SyncMap 而非普通 map 是为了保证并发安全，避免数据竞争。

3.2 核心函数实现

3.2.1 StartSnapshot 函数

该函数在服务器上启动快照过程:

```go
func (server *Server) StartSnapshot(snapshotId int) {
    // 记录本地状态
    snapshot := &SnapshotState{
        id:       snapshotId,
        tokens:   make(map[string]int),
        messages: make([]*SnapshotMessage, 0),
    }
    snapshot.tokens[server.Id] = server.Tokens
    server.snapshots.Store(snapshotId, snapshot)

    // 初始化marker追踪
```

```
    markers := make(map[string]bool)
    server.markersReceived.Store(snapshotId, markers)

    // 向所有出站链路发送marker
    server.SendToNeighbors(MarkerMessage{snapshotId})

    // 检查是否立即完成（无入站链路）
    if len(server.inboundLinks) == 0 {
        server.sim.NotifySnapshotComplete(server.Id, snapshotId)
    }
}
```

3.2.2 HandlePacket 函数

该函数处理接收到的消息：

```
func (server *Server) HandlePacket(src string, message interface{}) {
    switch msg := message.(type) {
    case TokenMessage:
        server.Tokens += msg.numTokens
        // 记录尚未收到marker的通道上的消息
        server.markersReceived.Range(func(key, value interface{})
            bool {
            snapshotId := key.(int)
            markers := value.(map[string]bool)
            if !markers[src] {
                snapVal, _ := server.snapshots.Load(snapshotId)
                snapshot := snapVal.(*SnapshotState)
                snapshot.messages = append(snapshot.messages,
                    &SnapshotMessage{src, server.Id, msg})
            }
            return true
        })
    case MarkerMessage:
        snapshotId := msg.snapshotId
        if _, exists := server.markersReceived.Load(snapshotId); !
            exists {
            server.StartSnapshot(snapshotId)  // 首次收到marker
        }
        markersVal, _ := server.markersReceived.Load(snapshotId)
```

```
        markers := markersVal.(map[string]bool)
        markers[src] = true

        // 检查是否所有入站链路都收到marker
        if len(markers) == len(server.inboundLinks) {
            server.sim.NotifySnapshotComplete(server.Id, snapshotId)
        }
    }
}
```

3.3 并发控制

由于分布式系统中存在多个并发的快照过程和消息传递，必须确保线程安全。本实现采用了 SyncMap 来保护共享数据结构，避免了 data race 问题。

4 测试结果

实现的算法通过了所有测试用例，包括：

- Test2NodesSimple: 两节点简单拓扑
- Test2NodesSingleMessage: 两节点单消息传递
- Test3NodesMultipleMessages: 三节点多消息传递
- Test3NodesMultipleBidirectionalMessages: 三节点双向消息传递
- Test8NodesSequentialSnapshots: 八节点顺序快照
- Test8NodesConcurrentSnapshots: 八节点并发快照
- Test10NodesDirectedEdges: 十节点有向边拓扑

5 总结

本次实验成功实现了 Chandy-Lamport 分布式快照算法，深入理解了分布式系统中全局状态捕获的原理和实现细节。通过处理并发控制、消息传递、状态记录等问题，掌握了分布式算法实现的关键技术。该算法在分布式系统的检查点、故障恢复、调试等场景中有重要应用价值。

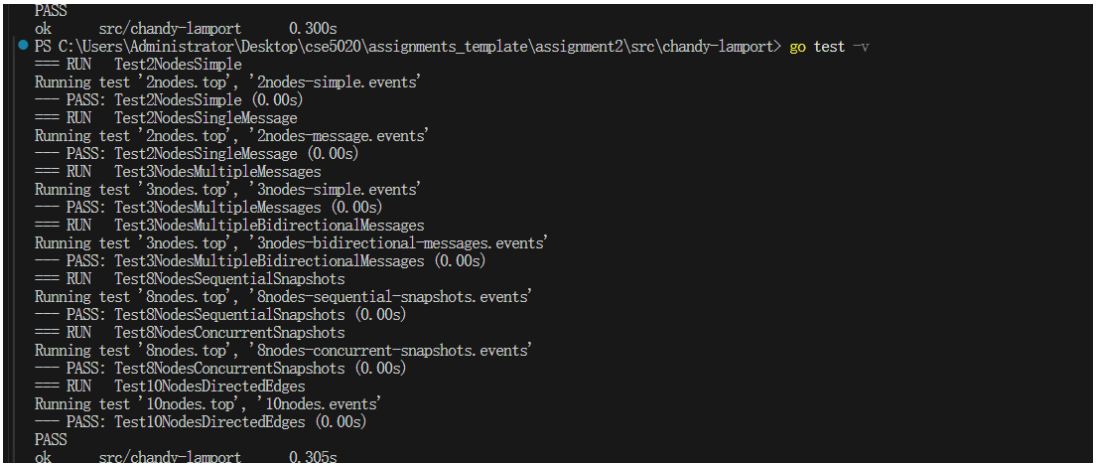

图 1: 所有测试用例通过的结果截图