# SYSTÈME MAPREDUCE DISTRIBUÉ AVEC DASHBOARD WEB

**Préparé par :**
Aymen Jeddou
Abderrahim Neji
Mohamed Amine Trabelsi

**Établissement :**
Faculté des Sciences de Bizerte
1ère année cycle ingénieur

**Date :** 20/05/2025

# TABLE DES MATIÈRES

# Remerciements

Nous tenons à remercier l'ensemble des enseignants du département d'informatique de la Faculté des Sciences de Bizerte surtout Mr. Khaled Barbaria pour la qualité des enseignements dispensés tout au long de l'année. Ce projet a été une occasion précieuse pour mettre en pratique les notions théoriques acquises, notamment en applications réparties.

# Introduction

Ce rapport présente notre implémentation d'un système MapReduce distribué avec un tableau de bord web pour le monitoring en temps réel. Le projet a été réalisé dans le cadre du cours d'Architectures réparties du cycle ingénieur à la Faculté des Sciences de Bizerte.

Le paradigme MapReduce, popularisé par Google, permet de traiter efficacement de grands volumes de données en parallélisant les calculs sur plusieurs machines. Notre implémentation, entièrement codée en Go, comprend un système distribué avec un master et plusieurs workers, ainsi qu'une interface web pour visualiser l'état du système en temps réel.

# Contexte et objectifs

L'objectif principal de ce projet était de concevoir et implémenter un système distribué de type MapReduce en Go, incluant un tableau de bord web pour le monitoring en temps réel des tâches et des workers. La solution devait être développée sans utiliser de framework externe pour la gestion RPC ou HTTP, en s'appuyant uniquement sur les bibliothèques standards de Go.

Les fonctionnalités principales à implémenter étaient : - La logique map/reduce en local - La communication RPC entre master et workers - L'intégration d'un tableau de bord web

Le système devait également être robuste face aux crashs et lenteurs des workers, avec une gestion appropriée des timeouts et des réattributions de tâches.

# Architecture du système

## Vue d'ensemble

Notre système MapReduce distribué est composé de trois composants principaux :

1. **Master** : Coordonne l'ensemble du processus, distribue les tâches aux workers, gère les timeouts et les réattributions, et expose une interface web pour le monitoring.

2. **Workers** : Exécutent les tâches map ou reduce assignées par le master, et rapportent leur achèvement.

3. **Dashboard Web** : Interface utilisateur permettant de visualiser l'état du système, les tâches en cours et les workers actifs.

Le processus MapReduce se déroule en trois phases principales : 1. **Phase Map** : Le master distribue les tâches de type map aux workers disponibles. 2. **Phase Reduce** : Après la fin de la phase map, le master lance les tâches reduce. 3. **Phase Merge** : Le master concatène les fichiers résultants pour produire le résultat final.

## Structure des fichiers

Notre projet est organisé selon la structure suivante :

```
PDIST25-AYMENJEDDOU-ABDERRAHIMNEJI-MOHAMEDAMINETRABELSI/
├── cmd/
│   ├── master/
│   │   └── main.go       # Point d'entrée pour le master
│   └── worker/
│       └── main.go       # Point d'entrée pour le worker
├── input/                # Fichiers d'entrée pour les tests
├── mapreduce/
│   ├── common.go         # Fonctions et constantes communes
│   ├── mapreduce.go      # Implémentation des fonctions DoMap
et DoReduce
│   ├── master.go         # Implémentation du master
│   ├── worker.go         # Implémentation du worker
│   └── word_count.go     # Fonctions spécifiques pour le
comptage de mots
├── tests/                # Tests unitaires
├── web/
│   └── index.html        # Interface web du dashboard
├── README.md             # Instructions de lancement du projet
└── rapport.pdf           # Rapport technique du projet
```

Cette organisation modulaire permet une séparation claire entre le master, les workers et le dashboard web, facilitant ainsi la maintenance et l'évolution du code.

# Implémentation du système MapReduce

## Fonctions ajoutées et complétées

À partir du code séquentiel fourni dans `sequential.zip`, nous avons ajouté et complété plusieurs fonctions pour implémenter le système distribué. Voici les principales modifications :

### Complétion de la fonction `DoReduce`

La fonction `DoReduce` a été complétée pour traiter les fichiers intermédiaires générés par les tâches map :

```
func DoReduce(
    jobName string,
    reduceTaskNumber int,
    nMap int,
    reduceF func(key string, values []string) string,
) {
    // Map de regroupement des valeurs par clé
    keyValues := make(map[string][]string)

    // Lire chaque fichier intermédiaire produit par les tâches
Map
    for i := 0; i < nMap; i++ {
        fileName := ReduceName(jobName, i, reduceTaskNumber)
        // Ouvrir le fichier pour la tâche de mappage i
        file, err := os.Open(fileName)
        if err != nil {
            panic("Erreur ouverture fichier intermédiaire: " +
err.Error())
        }
        // Lire les paires clé-valeur du fichier
        // Ajouter la valeur à la liste de valeurs pour cette
clé
        decoder := json.NewDecoder(file)
        var kv KeyValue
        for decoder.Decode(&kv) == nil {
            keyValues[kv.Key] = append(keyValues[kv.Key],
kv.Value)
        }
        file.Close()
    }
```

```
    // Récupérer les clés triées pour un ordre déterministe
    var keys []string
    for k := range keyValues {
        keys = append(keys, k)
    }
    sort.Strings(keys)

    // Ouvrir le fichier de sortie pour la tâche de réduction
    // utiliser MergeName
    outputFile, err := os.Create(MergeName(jobName,
reduceTaskNumber))
    if err != nil {
        panic("Erreur création fichier résultat reduce: " +
err.Error())
    }
    defer outputFile.Close()

    // Créer un encodeur JSON pour le fichier de sortie
    enc := json.NewEncoder(outputFile)

    // Réduire chaque clé et écrire le résultat
    // Appliquer la fonction de réduction à chaque clé
    for _, key := range keys {
        // Appliquer reduceF pour réduire les valeurs associées
à cette clé
        // Écrire la clé et la valeur réduite dans le fichier de
sortie
        reducedValue := reduceF(key, keyValues[key])
        err := enc.Encode(&KeyValue{Key: key, Value:
reducedValue})
        if err != nil {
            panic("Erreur encodage résultat reduce: " +
err.Error())
        }
    }
}
```

**Ajout de la structure `Master`**

Nous avons implémenté la structure `Master` pour gérer les tâches et les workers :

```
// Master gere les tasks et les workers
type Master struct {
    tasks      []Task
    workers    map[string]*WorkerInfo
    nReduce    int
    jobName    string
    files      []string
    mu         sync.Mutex
    done       chan bool
```

```
    tasksDone  int
    totalTasks int
}
```

Cette structure contient : - Une liste des tâches à exécuter - Une map des workers disponibles - Des informations sur le job en cours - Un mutex pour la synchronisation - Un canal pour signaler la fin du job - Des compteurs pour suivre l'avancement

### Ajout de la structure `Worker`

Nous avons également implémenté la structure `Worker` pour exécuter les tâches map et reduce :

```
// Worker execute map ou reduce
type Worker struct {
    id         string
    masterAddr string
}
```

Le worker communique avec le master via RPC pour demander des tâches et signaler leur achèvement.

### Implémentation des méthodes RPC

Nous avons ajouté deux méthodes RPC principales pour la communication entre le master et les workers :

1. **GetTask** : Permet à un worker de demander une tâche au master.
2. **ReportTaskDone** : Permet à un worker de signaler l'achèvement d'une tâche.

```
func (m *Master) GetTask(args *GetTaskArgs, reply
*GetTaskReply) error {
    m.mu.Lock()
    defer m.mu.Unlock()

    // Register worker if new
    if _, exists := m.workers[args.WorkerID]; !exists {
        m.workers[args.WorkerID] = &WorkerInfo{
            ID:      args.WorkerID,
            Status:  "idle",
            Address: args.WorkerID,
        }
    }

    // Find a pending or timed-out task
    now := time.Now()
```

```
    for i, task := range m.tasks {
        if task.Status == "pending" || (task.Status ==
"running" && now.Sub(task.StartTime) > 10*time.Second) {
            m.tasks[i].Status = "running"
            m.tasks[i].WorkerID = args.WorkerID
            m.tasks[i].StartTime = now
            reply.Task = task
            m.workers[args.WorkerID].Status = "working"
            Debug("Master: Assigned task %d (%s) to worker
%s\n", task.ID, task.Type, args.WorkerID)
            return nil
        }
    }

    // No tasks available
    reply.Task = Task{Type: IdleTask}
    m.workers[args.WorkerID].Status = "idle"
    Debug("Master: No tasks for worker %s, assigned idle\n",
args.WorkerID)
    return nil
}
```

### Communication RPC

La communication entre le master et les workers est réalisée à l'aide du package RPC standard de Go. Le master expose les méthodes RPC via un serveur HTTP :

```
func (m *Master) startRPC() {
    rpc.Register(m)
    rpc.HandleHTTP()
    listener, err := net.Listen("tcp", ":1234")
    CheckError(err, "cannot start RPC server: %v\n", err)
    go http.Serve(listener, nil)
}
```

Les workers se connectent au master via ce serveur RPC pour demander des tâches et signaler leur achèvement :

```
func (w *Worker) Run() {
    for {
        // Request task
        client, err := rpc.DialHTTP("tcp", w.masterAddr)
        if err != nil {
            Debug("Worker %s: Cannot connect to master: %v\n",
w.id, err)
            time.Sleep(time.Second)
            continue
        }
```

```
        var reply GetTaskReply
        err = client.Call("Master.GetTask",
&GetTaskArgs{WorkerID: w.id}, &reply)
        client.Close()

        // ... exécution de la tâche ...

        // Report completion
        client, err = rpc.DialHTTP("tcp", w.masterAddr)
        if err != nil {
            Debug("Worker %s: Cannot report task %d: %v\n",
w.id, reply.Task.ID, err)
            continue
        }
        var doneReply ReportTaskDoneReply
        err = client.Call("Master.ReportTaskDone",
&ReportTaskDoneArgs{TaskID: reply.Task.ID, WorkerID: w.id},
&doneReply)
        client.Close()
    }
}
```

### Gestion des tâches

Le master gère l'attribution des tâches aux workers et le suivi de leur état. Les tâches peuvent être dans l'un des états suivants : "pending", "running" ou "completed".

Lorsqu'un worker demande une tâche, le master lui attribue une tâche en attente ou une tâche dont le délai d'exécution a expiré. Cette approche permet de gérer les workers lents ou en panne.

### Tolérance aux pannes

Notre système est conçu pour être robuste face aux crashs et aux lenteurs des workers. Nous avons implémenté deux mécanismes principaux :

1. **Détection de timeout** : Si un worker ne termine pas sa tâche dans un délai de 10 secondes, la tâche est considérée comme abandonnée et peut être réattribuée à un autre worker.

2. **Simulation de pannes** : Pour tester la robustesse du système, les workers simulent aléatoirement des crashs (5% de probabilité) ou des retards (10% de probabilité) :

```
// Simulate crash (5%) or delay (10%)
if rand.Float64() < 0.05 {
    Debug("Worker %s: Simulating crash for task %d\n", w.id,
reply.Task.ID)
```

```
        os.Exit(1)
    }
    if rand.Float64() < 0.1 {
        Debug("Worker %s: Simulating delay for task %d\n", w.id,
    reply.Task.ID)
        time.Sleep(5 * time.Second)
    }
```

# Dashboard Web

## Interface utilisateur

Le dashboard web fournit une interface utilisateur pour visualiser l'état du système MapReduce en temps réel. Il affiche :

1. Une barre de progression indiquant le pourcentage de tâches terminées.
2. Une table listant toutes les tâches avec leur état et le worker assigné.
3. Une table listant tous les workers avec leur état.

Voici quelques captures d'écran du dashboard en action :

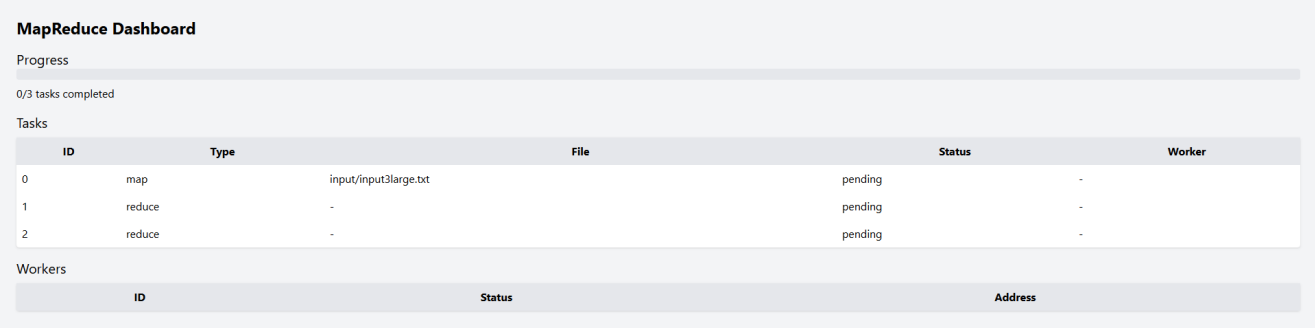


Figure 1: Dashboard initial avec les tâches en attente

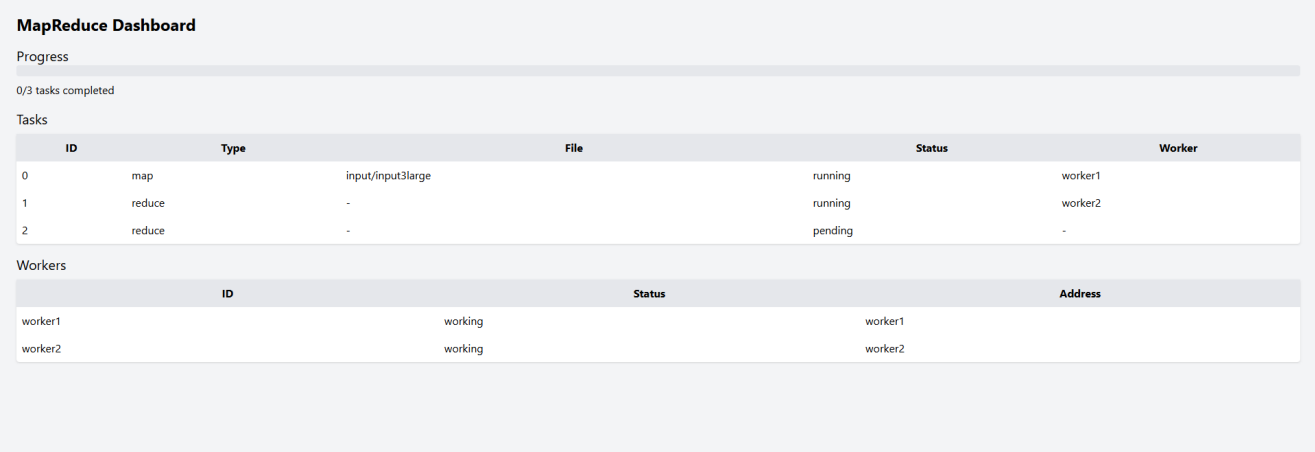


Figure 2: Dashboard pendant l'exécution des tâches

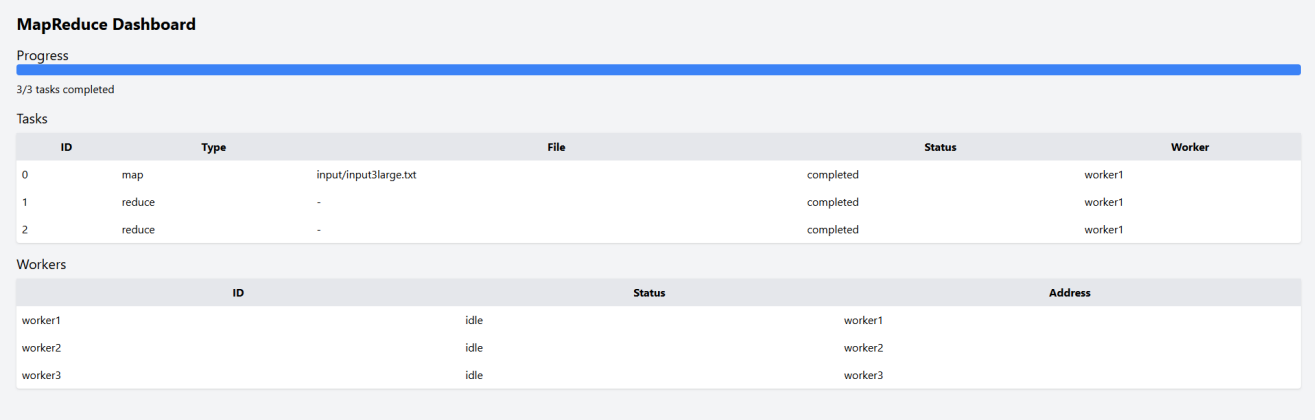


Figure 3: Dashboard après l'exécution de toutes les tâches

L'interface est implémentée en HTML et JavaScript, avec Tailwind CSS pour le style. Voici un extrait du code HTML :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>MapReduce Dashboard</title>
    <script src="https://cdn.tailwindcss.com"></script>
</head>
<body class="bg-gray-100 p-6">
    <h1 class="text-2xl font-bold mb-4">MapReduce Dashboard</h1>
    <div class="mb-4">
        <h2 class="text-xl">Progress</h2>
        <div class="w-full bg-gray-200 rounded">
            <div id="progress" class="bg-blue-500 h-4 rounded"
style="width: 0%"></div>
        </div>
        <p id="progress-text" class="mt-2"></p>
    </div>
    <!-- Tables for tasks and workers -->
    <!-- ... -->
</body>
</html>
```

## Communication avec le Master

Le dashboard web communique avec le master via une API HTTP. Le master expose deux routes principales :

1. / : Sert la page HTML du dashboard.
2. **/data** : Fournit les données en format JSON pour mettre à jour le dashboard.

```go
func (m *Master) startHTTP() {
    http.HandleFunc("/", func(w http.ResponseWriter, r
*http.Request) {
        http.ServeFile(w, r, filepath.Join("web", "index.html"))
    })
    http.HandleFunc("/data", m.serveData)
    go http.ListenAndServe(":8080", nil)
}

func (m *Master) serveData(w http.ResponseWriter, r
*http.Request) {
    m.mu.Lock()
    defer m.mu.Unlock()

    type Data struct {
        Tasks      []Task       `json:"tasks"`
        Workers    []WorkerInfo `json:"workers"`
        TasksDone  int          `json:"tasksDone"`
        TotalTasks int          `json:"totalTasks"`
    }

    data := Data{
        Tasks:      m.tasks,
        Workers:    make([]WorkerInfo, 0, len(m.workers)),
        TasksDone:  m.tasksDone,
        TotalTasks: m.totalTasks,
    }
    for _, worker := range m.workers {
        data.Workers = append(data.Workers, *worker)
    }
    json.NewEncoder(w).Encode(data)
}
```

Le JavaScript du dashboard interroge cette API toutes les secondes pour mettre à jour l'interface utilisateur :

```javascript
function updateDashboard() {
    fetch('/data')
        .then(response => response.json())
        .then(data => {
            // Update progress
            const progress = (data.tasksDone / data.totalTasks)
* 100;
            document.getElementById('progress').style.width =
progress + '%';
            document.getElementById('progress-
text').textContent = `${data.tasksDone}/$
{data.totalTasks} tasks completed`;
```

```
            // Update tasks and workers tables
            // ...
        })
        .catch(err => console.error('Error fetching data:',
err));
    setTimeout(updateDashboard, 1000);
}
updateDashboard();
```

## Tests et validation

Pour valider notre implémentation, nous avons effectué plusieurs tests :

1. **Tests unitaires** : Vérification du bon fonctionnement des fonctions `DoMap` et `DoReduce`.
2. **Tests d'intégration** : Vérification du bon fonctionnement du système complet avec un master et plusieurs workers.
3. **Tests de robustesse** : Vérification de la tolérance aux pannes en simulant des crashs et des retards de workers.

Les résultats des tests ont montré que notre système est capable de traiter efficacement des fichiers de différentes tailles, de gérer correctement les pannes de workers, et de fournir une interface web réactive pour le monitoring.

## Difficultés rencontrées et solutions

Au cours du développement de ce projet, nous avons rencontré plusieurs difficultés :

1. **Gestion des timeouts** : La détection des workers lents ou en panne a nécessité une gestion fine des timeouts. Nous avons résolu ce problème en enregistrant l'heure de début de chaque tâche et en vérifiant régulièrement si le délai d'exécution a été dépassé.

2. **Concurrence et synchronisation** : La gestion de l'accès concurrent aux structures de données partagées entre les goroutines a été un défi. Nous avons utilisé des mutex pour protéger ces structures et éviter les conditions de course.

3. **Débogage du système distribué** : Le débogage d'un système distribué est complexe car les erreurs peuvent survenir à différents endroits. Nous avons implémenté un système de journalisation détaillé pour faciliter le diagnostic des problèmes.

# Conclusion

Ce projet nous a permis de mettre en pratique des concepts avancés de systèmes distribués, de programmation concurrente et de développement web. Nous avons implémenté avec succès un système MapReduce distribué avec un tableau de bord web pour le monitoring en temps réel.

Notre implémentation respecte toutes les spécifications fonctionnelles proposées, notamment : - La logique map/reduce en local - La communication RPC entre master et workers - L'intégration d'un tableau de bord web - La robustesse face aux crashs et lenteurs des workers

Ce projet nous a également permis d'approfondir notre compréhension du langage Go et de ses bibliothèques standards pour le développement de systèmes distribués.

# Références


1. Documentation officielle de Go : https://golang.org/doc/
2. Go RPC Package : https://pkg.go.dev/net/rpc
3. Go HTTP Package : https://pkg.go.dev/net/http
4. Tailwind CSS : https://tailwindcss.com/docs
5. Tutoriel sur les systèmes distribués en Go : https://www.youtube.com/watch?v=UoB3j99RdP4
6. Blog sur l'implémentation de MapReduce : https://medium.com/@johnchourajr/implementing-mapreduce-in-go-f302820a6d0f
7. Cours MIT sur les systèmes distribués : https://www.youtube.com/playlist?list=PLrw6a1wE39_tb2fErI4-WkMbsvGQk9_UB
8. Tutoriel sur la programmation concurrente en Go : https://www.youtube.com/watch?v=f6kdp27TYZs
9. Article sur la tolérance aux pannes dans les systèmes distribués : https://www.infoq.com/articles/fault-tolerance-distributed-systems/
10. Tutoriel sur le développement d'interfaces web en Go : https://www.youtube.com/watch?v=SonwZ6MF5BE