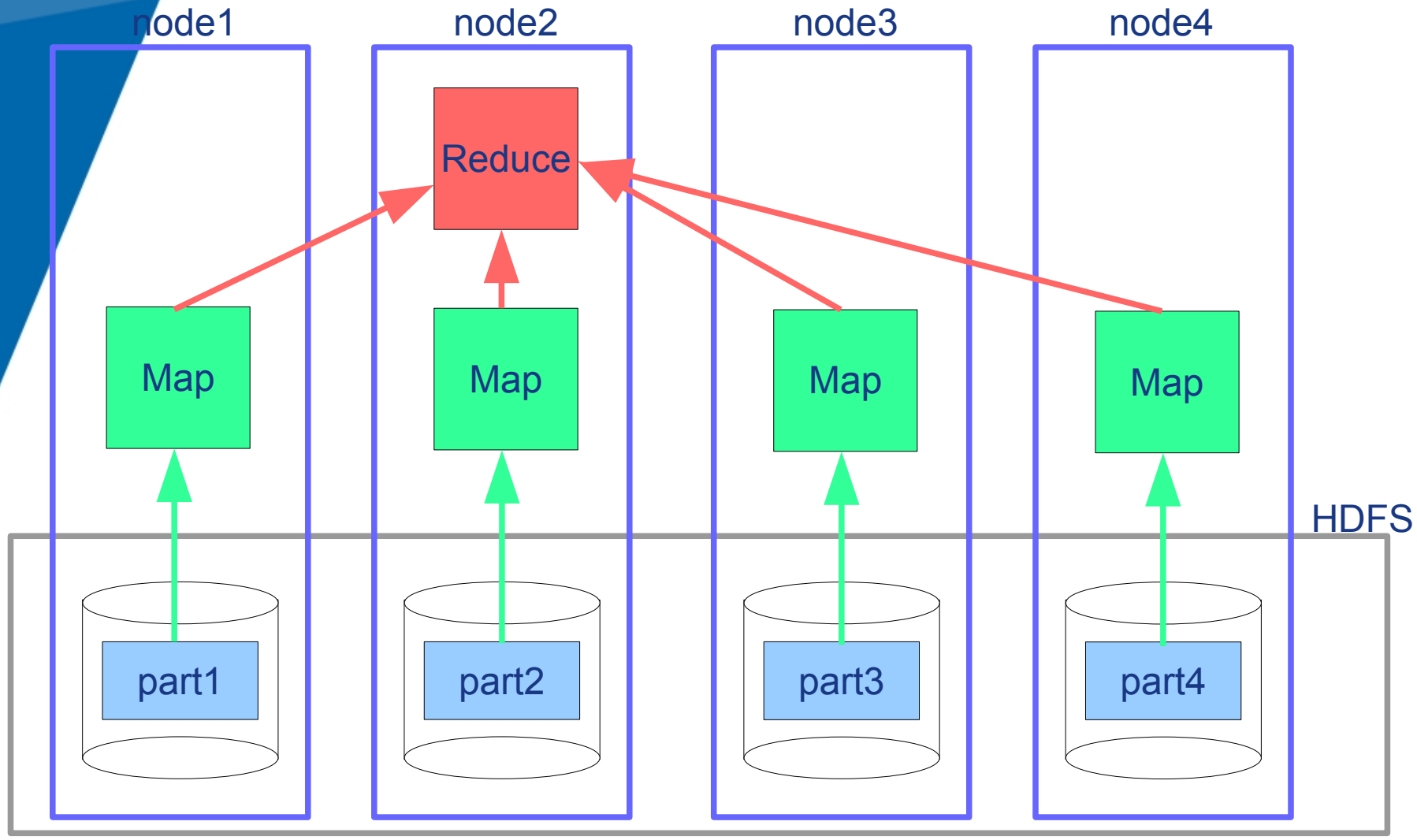
Projet Hidoop
Implantation d'une infrastructure de calcul map-reduce

Description Hidoop v0

Principe du map-reduce



Principe du map-reduce

- A noter que dans la v1
  - Il peut y avoir plusieurs Reduce
    - Une fonction shuffle permet de trier les résultats pour les envoyer au bon Reduce
  - Les fragments (parts) peuvent être dupliqués

Exemple d'application comptage de mots

- En itératif

```
HashMap<String,Integer> hm = new HashMap<String,Integer>();

// ouvrir fichier à lire : lnr
while (true) {
        String l = lnr.readLine();
        if (l == null) break;
        StringTokenizer st = new StringTokenizer(l);
        while (st.hasMoreTokens()) {
                String tok = st.nextToken();
                if (hm.containsKey(tok))
                        hm.put(tok, hm.get(tok).intValue()+1);
                else
                        hm.put(tok, 1);
        }
}

// recopier la hashmap dans le fichier résultat
```

Exemple d'application comptage de mots

- En map-reduce

```
public void map(FormatReader reader, FormatWriter writer) {

        HashMap<String,Integer> hm = new HashMap<String,Integer>();
        KV kv;
        while ((kv = reader.read()) != null) {
                StringTokenizer st = new StringTokenizer(kv.v);
                while (st.hasMoreTokens()) {
                        String tok = st.nextToken();
                        if (hm.containsKey(tok))
                                hm.put(tok, hm.get(tok).intValue()+1);
                        else
                                hm.put(tok, 1);
                }
        }
        for (String k : hm.keySet())
                writer.write(new KV(k,hm.get(k).toString()));
        }
}
```

Exemple d'application comptage de mots

- En map-reduce

```
public void reduce(FormatReader reader, FormatWriter writer) {

        HashMap<String,Integer> hm = new HashMap<String,Integer>();
        KV kv;
        while ((kv = reader.read()) != null) {
                if (hm.containsKey(kv.k))
                        hm.put(kv.k, hm.get(kv.k)+Integer.parseInt(kv.v));
                else
                        hm.put(kv.k, Integer.parseInt(kv.v));
        }
        for (String k : hm.keySet())
                writer.write(new KV(k,hm.get(k).toString()));
}
```

Lecture/écriture des données

- Dans un fichier du système de fichiers local
- Dans un fragment dans HDFS
- On gère des formats de données
  - On lit des données dans un format et on retourne une KV
  - On donne une KV et on écrit dans un format

```
public interface FormatReader {
        public KV read();
}
public interface FormatWriter {
        public void write(KV record);
}
```

Lecture/écriture des données

- Un format de fichier implante l'interface Format
- On gère deux format :
  - TxtFormat : une classe pour les fichiers texte
  - KVFormat : une classe pour des fichiers KV

```
public interface Format extends FormatReader, FormatWriter, Serializable {
        public enum Type { LINE, KV };
        public enum OpenMode { R, W };

        public void open(OpenMode mode);
        public void close();
        public long getIndex();
        public String getFname();
        public void setFname(String fname);

}
```

Des key-values

```
public class KV {
		public static final String SEPARATOR = "<->";
		public String k;
		public String v;

		public KV() {}

		public KV(String k, String v) {
			super();
			this.k = k;
			this.v = v;
		}

		public String toString() {
			return "KV [k=" + k + ", v=" + v + "]";
		}

}
```

HDFS

- Permet de gérer des fichiers fragmentés sur les nœuds
  - Quand on copie un fichier du FS local dans le FS HDFS, le fichier est coupé en fragments qui sont copiés sur les nœuds.
  - Quand on copie un fichier du FS HDFS dans le FS local, les fragments sont rassemblés pour obtenir le fichier complet sur le FS local.
- Les fragments sont copiés sur le FS local du nœud avec un nom particulier
- Les fragments sont de taille variable et non répliqués (v0)

HDFS

```
public class HdfsClient {
        public static void HdfsDelete(String hdfsFname) {…}
        public static void HdfsWrite(Format.Type fmt, String
                        localFSSourceFname,  int repFactor) {…}
        public static void HdfsRead(String hdfsFname,
                        String localFSDestFname) {…}
        public static void main(String[] args) {
                // java HdfsClient <read|write> <txt|kv> <file>
        }
}
```

Hidoop

- Interface d'un démon

```
public interface Daemon extends Remote {
        public void runMap (Mapper m, Format reader, Format writer, CallBack cb)
                        throws RemoteException;
}
```

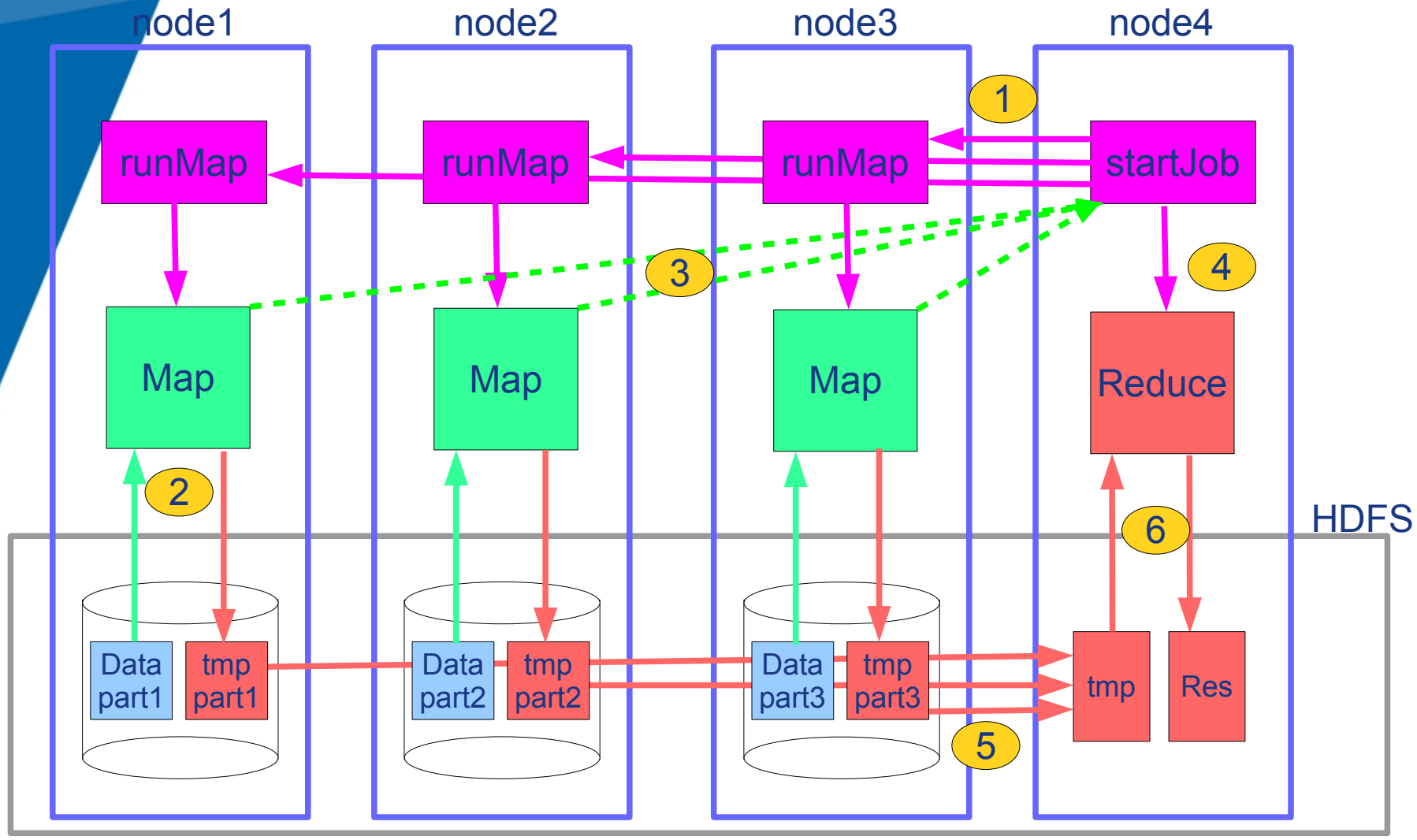
- Lancement d'un Job

```
public class class Job implements JobInterface {
                public void setInputFormat(Format.Type ft) {…}
                public void setInputFname(String fname)    {…}
                public static void startJob (MapReduce mr) {…}
}
```

- Extensions possibles pour JobInterface

```
public void setNumberOfMaps(int tasks);
public void setNumberOfReduces(int tasks);
public void setOutputFormat(Format.Type ft);
public void setOutputFname(String fname);
public void setSortComparator(SortComparator sc);
// et getXXX()
```

Hidoop


node1
node2
node3
node4
runMap
runMap
runMap
startJob
1
3
4
Map
Map
Map
Reduce
2
6
HDFS
Data part1
tmp part1
Data part2
tmp part2
Data part3
tmp part3
tmp
Res
5

1. startJob lance les Map en appelant runMap sur les démons
2. Les Map calculent en lisant localement un fragment et génèrent un fragment de résultat (fragment de tmp)
3. Les map appellent le CallBack pour dire qu'ils ont fini
4. startJob lance le reduce
5. Le Reduce copie dans son FS local le fichier tmp résultat des Map (composé de fragments), grâce à HdfsClient
6. Le Reduce calcule en lisant localement dans tmp et génère le fichier résultat final

Modèle de programmation

```
public interface Mapper extends Serializable {
        public void map(FormatReader reader, FormatWriter writer);
}
public interface Reducer extends Serializable {
        public void reduce(FormatReader reader, FormatWriter writer);
}
public interface MapReduce extends Mapper, Reducer {
}
```

- Les lectures/écritures se font toujours sur des fichiers ou fragments locaux

Quelques pistes pour la v1

- Faire un Hidoop tolérant les pannes
  - Les fragments sont répliquées dans HDFS
  - Hidoop sélectionne un ensemble de nœuds possédant les fragments requis, puis lance les calculs
  - Détection d'une panne d'un nœud (heartbeat), on relance les calculs
- Faire un Hidoop avec plusieurs Reduce
  - Indiquer le nombre de Reduce, Hidoop sélectionnant les nœuds pour les Reduce
  - Etendre les Format pour écrire les fragments de résultats à distance sur les nœuds des Reduce
  - Speedup si gros fichiers à réduire

Terminologie Hadoop

Fichiers

- *NameNode* : gère les métadonnées du SGF
- *DataNode* : mise en en œuvre des accès aux chunks sur un nœud

Exécution

- *RessourceManager* : gestion globale des ressources (allocation, supervision)
- *NodeManager* : gère les ressources sur un nœud (≈ démon)
- *ApplicationMaster* : supervise l'exécution pour une application donnée (≈ Job)