**Planning**

* Read into MapReduce, P2P, TomP2P
* Gather/install required Software
* Understanding code etc.
* Think about what libraries to use (TomP2P, IRIS for messaging)
  + Think about what to transfer (status messages, data)
  + Think about whole stack

**Splitting the work**

* **Implement simple MapReduce on TomP2P.** 
  + **Simple:** Be able to connect multiple computers, write a simple program in e.g. java script, deploy it with data to process, and be able to get the result of the reduce phase.
  + **MapReduce:**
    - **Submitting a Job:** Provide possibilities for both Java AND Javascript!
      * Needs a IJobSubmissionStrategy
    - **Distributing tasks**:
      * Directly over TomP2P?
      * Use IRIS?
      * Plan task distribution such that data transfer is minimized (like Hadoop). Also during reduce phase: concentrate data related to a given key in a single node and allocate the same node for reducing of such data
      * **Schedulers**:
        + **Hadoop:** Fair Scheduler, Capacity Scheduler, Task Scheduler (1.X), YarnScheduler (2.X)
    - **Storing data:**
      * Directly in the DHT
    - **Map:** 
      * split data in an appropriate way
        + simplest: use the files as M (in practice: mostly #tasks = #available nodes
        + already better: split according to number of available resources (e.g. computers)
        + even better: split according to number of available cpu’s per computer
        + etc. 🡪 goes into context awareness 🡪 need a IMapContextStrategy that allows for context-dependent splits
      * Process data according to user defined map-function
    - **Shuffle and Sort/ Combine, Partition:** 
      * aggregate, sort and combine results from map phase locally
        + **partition** if there are too many data items for each key (🡪 assign intermediate keys)
        + **combine** if there are too many keys
      * Data from map tasks are organized for reduce tasks
    - **Reduce:** aggregate and combine results from map phase globally
* **Implement Context-awareness**
  + **Gather** **information** about a node’s **configuration** & **current** **status** and **reason** about this information
    - **Context monitoring (platform independent)** for **scheduling, easily obtainable and lightweight** (no impact on application performance). **New nodes** may **join or leave at any time** without impacting monitoring. Configuration according to such context information needs to **automatically** be performed.
      * I think about implementing an IContextSubject that has List<NodeInformation implements IContextObserver> participatingNodesInformation and informs them when a new Node arrives or an old Node disappears or when information about a node is updated (notifyParticipants(NodeInformation nodeInformation))… Yeah something like that, don’t know yet…
    - **Context models**: context information needs to be represented in an **appropriate, extensible** model (e.g. simple key/value models vs. complex ontologies) 🡪 in MR application mostly important: **DATA LOCALITY** (adapting task distribution means adapt data allocation, too 🡪 not independent)
    - **Context distribution**: should be distributed on the grid, **no centralized approach**, **avoid communication flooding (network bandwidth is very important!!!!) 🡪 e.g. heartbeat messages to nodes (check if still alive), piggybacked over exchange messages 🡪 less context distribution messages on the network**