map/reduce (DIY)

#EmbarassinglyParallelPrograms #Make #SoftwareSelection #ProcessModeling #Redis #Windows

Embarassingly parallel processing with no setup

We have 100 pairs of big files to be processed by the same program (e.g. a full comparison)

→ *the problem is 'embarassingly parallel'*
The processing time is 6 minutes each.
→ the total processing time seems to be 10 hours.

We want to explore options to make the total time shorter.
we have a Windows domain with 25 idle machines.

Installing distributed-workflow software and agents on our machines can be complicated by long approval chains or skill shortage.

A possible solutions is based on a single Windows script that,

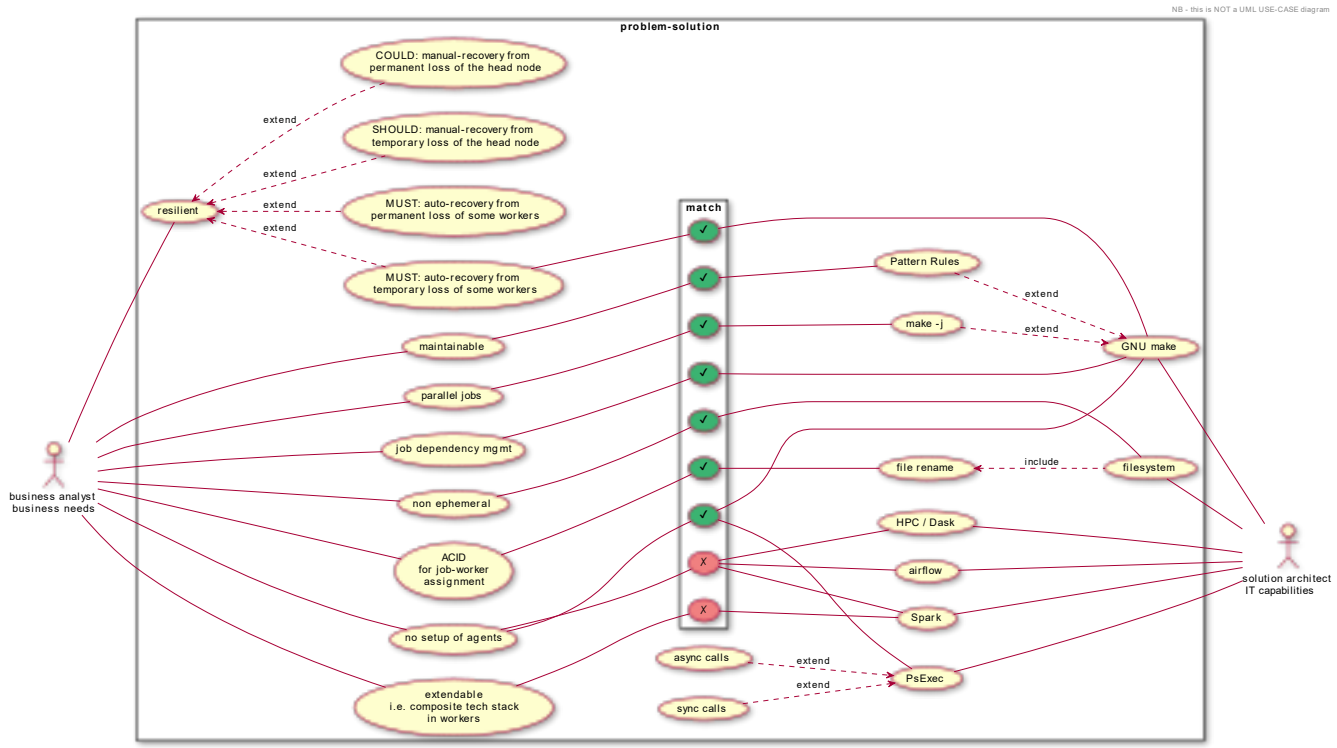
- with the help of PsExec.exe, submits remote executions on the 25 machines of the domain, and,
- with the help of the GNU make.exe, performs an automatic parallelization of the map tasks.

An alternative option may be based on Spark. A Spark-based project is detailed inside this website here.

The total execution time is expected to be close to 6 * 100/25 minutes = less than half hour (assuming the latency of file-transfers is negligible).

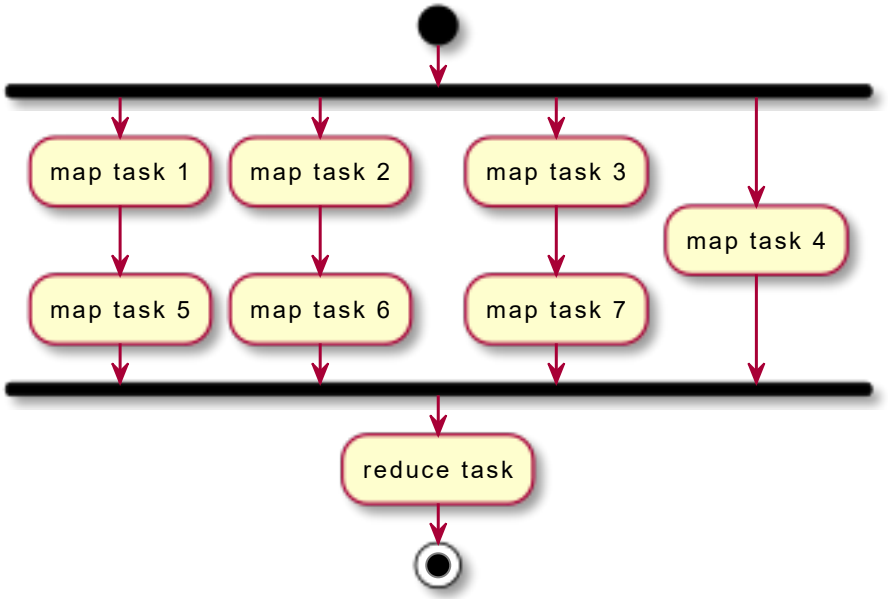
building the tech stack

The following diagram recaps our needs on the left, the tech opportunities on the right, what is adopted and what is ruled-out:

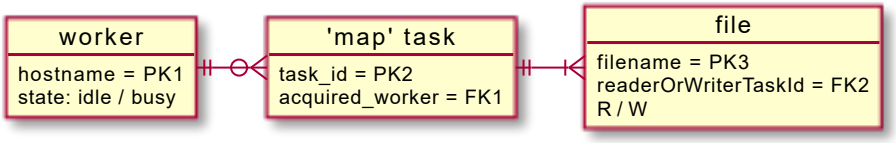


---

a map-reduce work schedule, assuming 7 map-tasks and 4 nodes:



E-R diagram of the entities used by the orchestrator (GNU make):

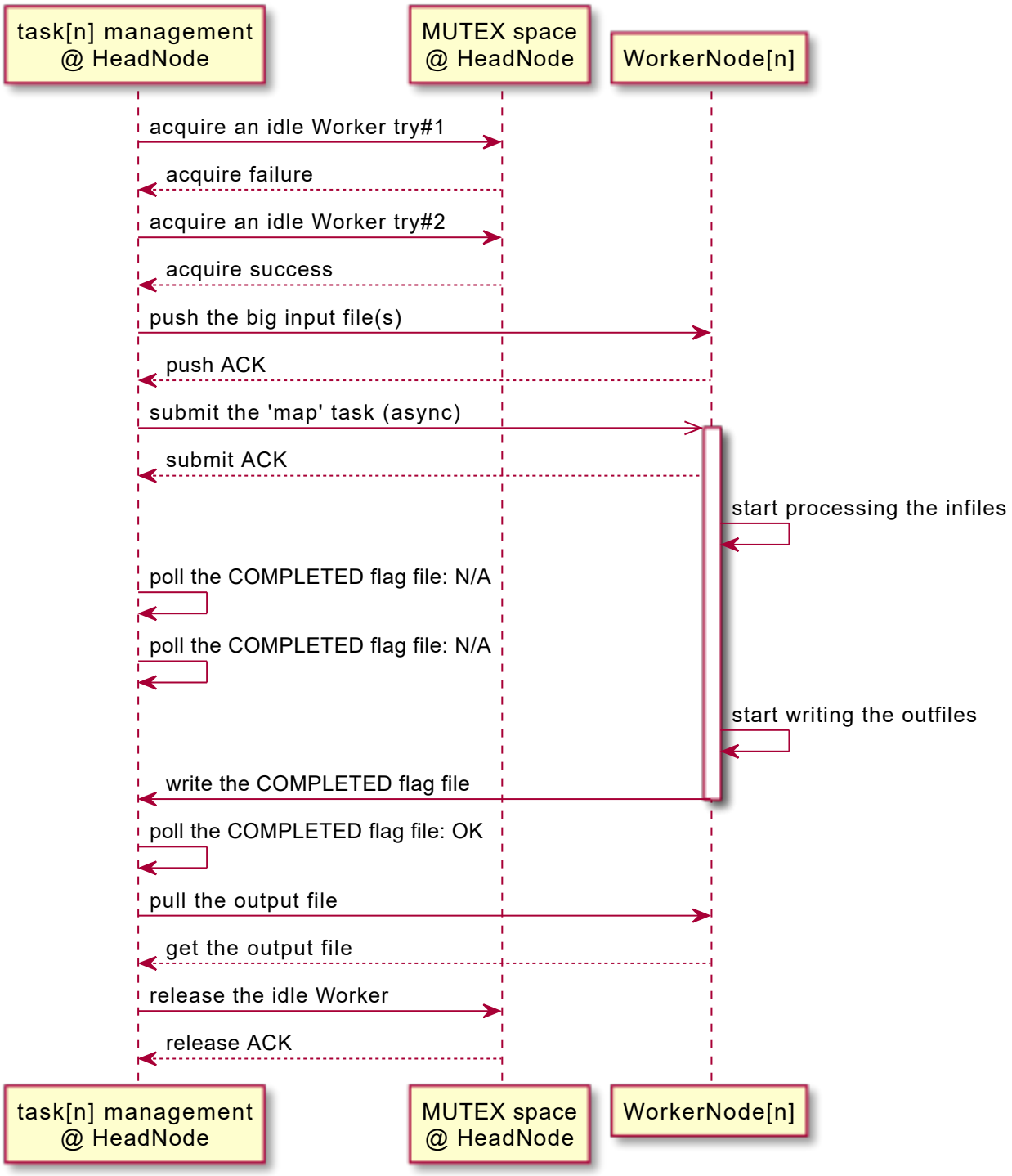


NB

- in such map-reduce scenarios, each file can either be input or output of one and only one task.
- we stick to max one task per worker at any given time.

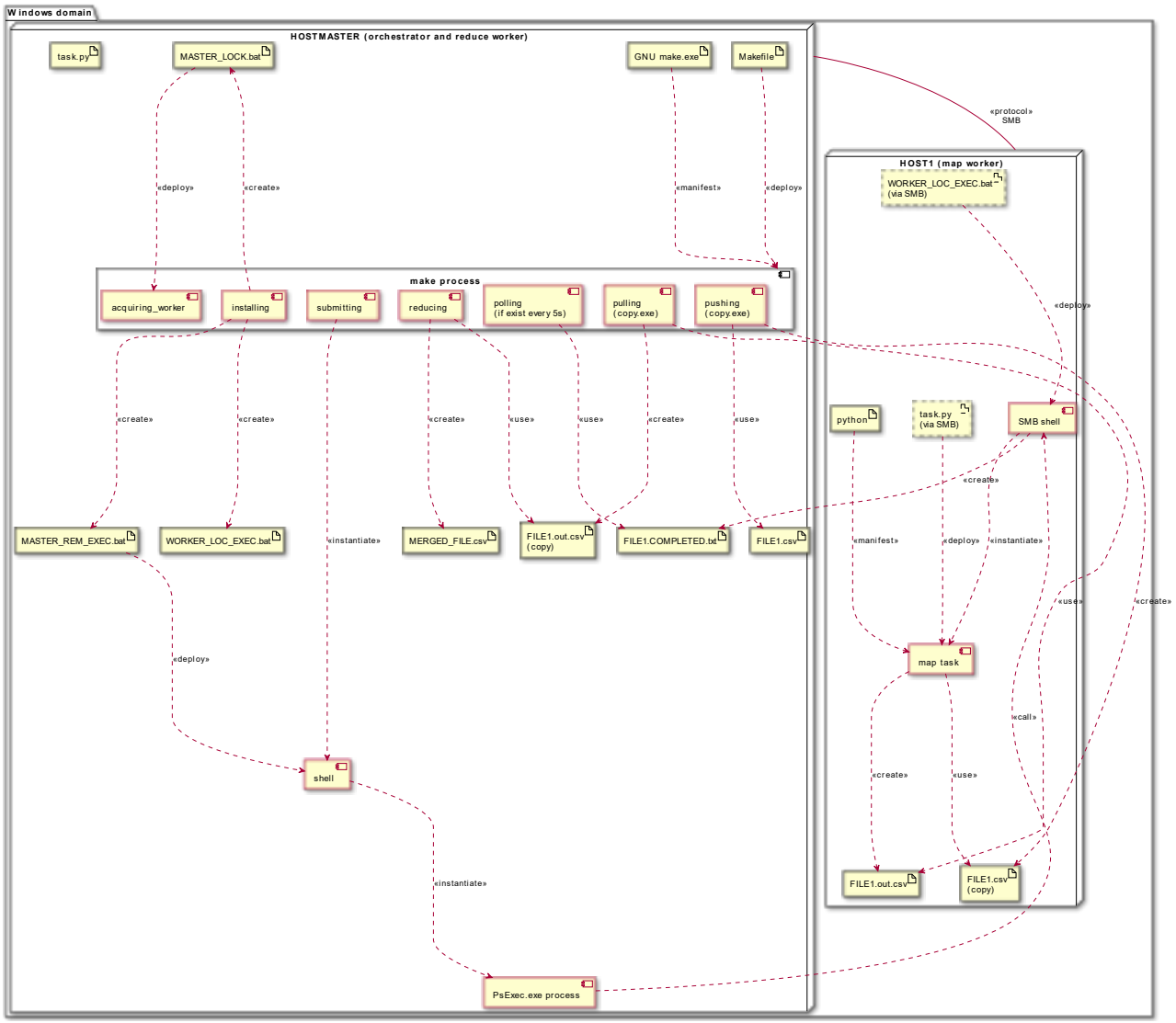
---

sequence diagram of the actions performed by each map task:



NB: again, nothing needs to be deployed on the workers machines, except for the input and output data files.

deployment diagram:



The Makefile that defines the steps and the dependencies of the whole process is available at this link: https://github.com/a-moscatelli/home/blob/main/am-wiki-assets/mapreducewin/Makefile

Based on the Makefile above, we just run:

```
set PARALLEL_DEGREE=3
make install
make initialize
make -j %PARALLEL_DEGREE% all
```

The activity log, with five 1-minute long map jobs and 3 nodes, is as below:

```
23:24:56.16 EXEC_ initialize

23:25:01.78 BEGIN TASK001.step1.workerAcquisitionLoopFinished.mkcontrol
23:25:01.84 BEGIN TASK002.step1.workerAcquisitionLoopFinished.mkcontrol
```

```
23:25:01.86 BEGIN TASK003.step1.workerAcquisitionLoopFinished.mkcontrol
23:25:05.97 END__ TASK001.step1.workerAcquisitionLoopFinished.mkcontrol
23:25:06.05 BEGIN TASK004.step1.workerAcquisitionLoopFinished.mkcontrol
23:25:06.11 END__ TASK002.step1.workerAcquisitionLoopFinished.mkcontrol
23:25:06.13 END__ TASK003.step1.workerAcquisitionLoopFinished.mkcontrol
23:25:06.19 BEGIN TASK005.step1.workerAcquisitionLoopFinished.mkcontrol
23:25:06.28 BEGIN TASK001.step2.inputFilePushedToWorker.mkcontrol HOST003
23:25:06.46 BEGIN TASK002.step2.inputFilePushedToWorker.mkcontrol HOST002
23:25:06.64 BEGIN TASK003.step2.inputFilePushedToWorker.mkcontrol HOST001
23:25:06.87 BEGIN TASK001.step3.submittedToWorker.mkcontrol HOST003
23:25:07.40 BEGIN TASK002.step3.submittedToWorker.mkcontrol HOST002
23:25:07.91 BEGIN TASK003.step3.submittedToWorker.mkcontrol HOST001

23:26:09.53 END__ TASK001.step4.workerCompletionCheckLoopFinished.mkcontrol
23:26:09.67 END__ TASK002.step4.workerCompletionCheckLoopFinished.mkcontrol
23:26:09.82 END__ TASK003.step4.workerCompletionCheckLoopFinished.mkcontrol
23:26:15.17 END__ TASK004.step1.workerAcquisitionLoopFinished.mkcontrol
23:26:15.28 BEGIN TASK004.step2.inputFilePushedToWorker.mkcontrol HOST003
23:26:15.30 END__ TASK005.step1.workerAcquisitionLoopFinished.mkcontrol
23:26:15.44 BEGIN TASK005.step2.inputFilePushedToWorker.mkcontrol HOST002
23:26:15.51 BEGIN TASK004.step3.submittedToWorker.mkcontrol HOST003
23:26:15.65 BEGIN TASK005.step3.submittedToWorker.mkcontrol HOST002

23:27:16.85 END__ TASK004.step4.workerCompletionCheckLoopFinished.mkcontrol
23:27:16.97 END__ TASK005.step4.workerCompletionCheckLoopFinished.mkcontrol
23:27:17.00 BEGIN reduce
23:27:17.08 END__ reduce
```

links:

- https://learn.microsoft.com/en-us/sysinternals/downloads/psexec
  the agent that makes remote calls
- https://www.gnu.org/software/make/manual/
  the agent that enables the parallel executions

the semaphore, required to ensure that only one process at a time can acquire a worker node, is based on the success/failure of a file rename.

Such option does not handle well the possibility that a lock holder will not survive the moment it is supposed to release the lock for other purposes.

a solution can be easily implemented using redis:

```
#docker
image: "redis:6.0.9"
```

Windows client:

https://github.com/microsoftarchive/redis/releases

example of a working script:

```
set REDISCLI=.\Redis-x64-3.0.504\redis-cli.exe
set REDISSERVERHOST=DESKTOP-B12345T
set REDIS_CALL=%REDISCLI% -h %REDISSERVERHOST% -p 6379

%REDISCLI% -h %REDISSERVERHOST% -p 6379 PING
rem > PONG
echo %ERRORLEVEL%
rem > 0

set autoexpire_seconds=600

set key=HOST1
set val=TASK4

rem simulation of a lock acquisition"
%REDIS_CALL% SET %key% %val% EX %autoexpire_seconds% NX
rem > OK

rem simulation of a competing concurrent lock acquisition:
%REDIS_CALL% SET %key% %val% EX %autoexpire_seconds% NX
rem > (nil)

%REDIS_CALL% KEYS "*"
rem > 1) "HOST1"

rem simulation of a lock release:
%REDIS_CALL% DEL %key%
rem > (integer) 1

%REDIS_CALL% KEYS "*"
rem > (empty list or set)
```

%ERRORLEVEL% after each CLI call above is always 0

---

back to Portfolio

---

backlog 🔒