COS 786: The Pirate Project

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Date Issued: **13 February 2017**
- Date Due: **28 May 2017**
- Date Demo: **29 May - 2 June 2017**

Contents

1 Introduction

During this project you will implement a distributed system that utilizes a group of individual agents, each solving smaller tasks, in order to accomplish a common goal. You will learn how to implement stand-alone agents, how to elect a leader amongst these agents, how to efficiently communicate between processes, and how to implement a fault-tolerant system.

2 Storyline

Captain Rummy Rum, the greatest pirate captain of all time, has overheard whispers of a treasure hidden on a remote island. Captain Rummy decides to set out for Port Noroyal to hire a crew of pirates to help him find this legendary treasure. He has gathered a number of clues indicating different islands that may possibly have the buried treasure. Since Captain Rummy is already 99 years old and wants to spend as much as he can before he dies, he has to find the treasure as quickly as possible. Captain Rummy also wants to appoint a quartermaster from his crew that will coordinate the search in order to reduce the time required to visit each of the islands and determine if the treasure is hidden there. As pirates are, some of the crew members want the treasure all to themselves and will try to cheat the captain and the rest of the crew by lying or providing misinformation. However, Captain Rummy was not born yesterday, has many years experience with crooked pirates, and can easily detect lies. The Captain may decide to get rid of these corrupt pirates by simply throwing them overboard. However, replacing these pirates is time-consuming, since they have to sail back to a port to recruit new crew members. Hence, Captain Rummy may decide to (a) keep the corrupt pirates with the hope they might better themselves, (b) throw them overboard and continue the search without the additional help, or (c) replace them by going back to a port and then continuing the search with a full crew. The main objective of Captain Rummy is to find the treasure. The secondary objective is to find the treasure as efficiently and as quickly as possible.

3 Distribute System

You are given an existing program, `rummy`, representing Captain Rummy. This is a terminal-based program accepting JSON inputs and providing JSON outputs.

3.1 Communication

You are required to program stand-alone *agents*, where each a agent represents a single crew member. These agents are individual executables/scripts which are only allowed to communicate with each other and with the captain through an external mechanism. Hence, you are not allowed to create a single program and have individual objects/functions representing the pirates and communicate through function calls, that is **not** a distributed system. If you run multiple agents, each of these agents should show up individually on the OS task/system monitor. Some external communication mechanisms that you may want to use are:

- Files: Individual agents read/write to a shared file. You have to provide locking mechanism to avoid collisions during simultaneous writes.
- Database: Individual agents read/write to a shared database. Most databses have a built-in atomic operations to avoid collisions during simultaneous writes.
- Sockets: Use sockets to communicate between processes. This requires some standard and format to send commands and data across the socket.
- RPC Mechanisms: There are numerous available libraries that utilize sockets and HTTP to facilitate remote procedure calls via XML, JSON, or other formats.
- MPI Mechanisms: There are numerous available libraries that provide a message passing interface.
- Other Mechanism: There are numerous other mechanism for process communication that will be discussed during the semester.
- Custom Mechanism: Create your own custom RPC, MPI, or other mechanism to communicate between processes.

These communications mechanisms all have different locking and message throughput, which will have a major impact on the overall performance of your system. Students will be evaluated based on their choice of communication. Advanced RPC, MPI, and other mechanisms will earn some additional marks, and custom mechanisms will earn even more marks. Choose wisely.

3.2 Leadership

You will have to provide an interface to the provided program in order to facilitate communication between the Captain, the quartermaster and the crew members. You may choose to employ a leader or let every agent work on his own. The following two options are available:

- Central Leadership: Create a quartermaster agent who is the only one that communicates with the captain. All the crew members have to speak to the quartermaster, which in turn will speak to the captain. The crew members are never allowed to communicate with the captain.
- No Leadership: No quartermaster is employed and all pirates talk directly to the captain. You will have to rely on the throughput of the Captain, that is the `rummy` program, and he is lazy and slow.

Implementing leadership between the agents will earn additional marks.

3.3 Corruption

Some of the pirates are corrupt and want to keep the treasure to themselves. Similar, in a distributed system, there may be malicious agents, corrupt network communication, or incorrect calculations. To simulate these problems, the Captain will hand out clues to the individual pirates. Each pirate has a random probability of being corrupt, which is determined the moment the pirate is added to the crew, and does not change over the lifetime of the pirate. When the Captain hands out clues to the crew, some of the clue data is corrupt, based on the corruption probability of the receiving pirate. Hence, if a non-corrupt clue is solved, the correct key is produced. On the other hand, if a corrupt clue is solved, the incorrect key is produced. The Captain knowns whether or not the key is valid, and if not, will instruct the pirate to solve it again with new clue data (which might or might not be corrupt again). You have to handle these corrupt agents in some manner:

- Ignore: You may simply ignore corrupt agents. Depending on the probability, some agents might generate a lot of incorrect keys which then have to be resolved. This creates a major computational overhead, but does not require any additional code.
- Remove: If an agent is too corrupt, you can simply remove him and put the workload on the remaining agents. This requires only some additional code, but if too many pirates are thrown overboard, your remaining agents may be overused and since they run on a single thread (see later), your CPU will not be fully utilized.
- Replace: You can throw corrupt pirates overboard and hire new ones. You thus keep a full crew all the time and can maximize your CPU utilization. Although removing an agent is cheap, adding a new one is costly and should be done sparingly. You may for instance decide to only add agents at the start of the program and not at the end when only one or two clues are left.

If you decide to remove agents, you have to figure out exactly when to remove them. There are numerous algorithms to determine fault detection, although a lot of them do not apply to this "simple" situation. The complexity and efficiency of detecting, removing, and adding agents will have an influence on your marks.

3.4 Networking

Since most of you do not have access to multiple computers, you are not required to implement a "true" distributed system that can run over the network. You can just run all your agents on a single machine. However, implementing your agents to both work locally and over the network is not difficult if you use databases, sockets, RPCs, or MPIs. You can test your implementation locally, use a few virtual machines, or got the labs and make everyone angry by occupying and entire row of computers. You will earn extra marks if your system is indeed distributed and can run over the network.

4 Environment

This section provides some details on the environment and programming language restrictions.

4.1 Programming Language

You may use any programming language you like. However, it is **strongly** advised to make use of Python, since the main program, `rummy`, is also written in Python and provides easy integration. Additionally, Python has numerous easy-to-use integrated libraries that will make your life a lot easier. Python has everything, from hashing functions, to database providers and sockets.

If you choose to use Python, please either use version 2.7.13 or 3.6.0. The provided program, `rummy`, is precompiled and can only be used with specific versions of Python. Most other Python version should work as well, just try and see. You can use your native Linux package installer or download Python for any platform here:

- Python 2.7.13: https://www.python.org/ftp/python/2.7.13/
- Python 3.6.0: https://www.python.org/ftp/python/3.6.0/

If you choose to use another language, there are some additional restrictions discussed below. You will also have to submit makefiles and instructions on how to compile and run the program. The final project will be tested on a Linux machine. If your code does not compile and run, you will get zero for the project.

4.2 Operating System

You may use any operating system you like. However, it is **strongly** advised to make use of Linux, because reasons. Your final project will be tested on a Linux (Ubuntu 16.10 64bit) machine. Hence, you may **not** utilize any platform-specific libraries or function calls. If you use Python, the code is cross-platform, and your code will run on Linux, even if you used Windows or Mac. Just make sure you do not include something platform-specific, such as `win32` or `darwin`.

4.3 Provided Files

The provided program, `rummy`, is precompiled to prevent students from tampering with the core code and to provide a learning experience with precompiled tools which are often used in distributed systems. All program data is fully encrypted, so do not try to circumvent `rummy`, only the Captain knows the clues.

If you use Python, there are two versions (2.7.13 and 3.6.0) available which should work on any operating system. If you use a different Python version, just try to run it, it will most likely work. If the code is incompatible, you will get a `bad magic number` error. In order to run the program, execute the following command:

```
>> python rummy.pyc
```

On Windows, in order to execute the command, you may need to add the Python installation directory to your environment variables for versions lower than Python 3. Here are some instructions on how to do it:

http://windowsitpro.com/systems-management/how-can-i-add-new-folder-my-system-path

`rummy` requires the Python PIL library to read the provided input images. On most Linux systems this should be installed by default. If not, or if you use Windows, run the following command:

```
>> pip install image
```

On Windows, the PIP program is located in `<Python Install Directory>/Scripts/pip.exe`.

If you use any other programming language, you can still use the Python program for your interface, as long as you have Python installed. If you do not wish to install Python or the PIL library, there is also a precompiled executable for Windows and Ubuntu (should also work on other Linux distros). On Windows, execute:

```
>> rummy.exe
```

and on Linux:

```
>> rummy
```

These executables should run without any additional decencies or required installations.

The following files are provided to students:

1. `COS786_Project_Python.zip`: The executable (both version 2.7 and 3.6) if you have Python installed. Works on all operating systems.
2. `COS786_Project_Linux.zip`: The executable if you work on Linux and do not want to use Python.
3. `COS786_Project_Windows.zip`: The executable if you work on Windows and do not want to use Python.
4. `COS786_Project_Data.zip`: The image maps and the pregenerated clues. These files must be copied to your executable `data` directory.
5. `COS786_Project_Specifications.pdf`: This document.

4.4 Inputs and Outputs

All inputs to and outputs from `rummy` are in JSON format. Python and most other programming languages provide native support for JSON parsing. If you do not know anything about JSON, here is a quick tutorial, should take you no more than 10 minutes:

http://www.w3resource.com/JSON/introduction.php

During the project you may get huge lists of data, making it difficult to read and interpret. You can use JSON prettifiers to make them more readable:

https://jsonformatter.curiousconcept.com

Note that you are interacting with a terminal program. All parameters send to `rummy` **must** be in quotes. For instance, if you have a JSON list `[1, 2, 3]`, you must pass it to `rummy` as `"[1, 2, 3]"` or `'[1, 2, 3]'`.

4.5 Multithreading

COS786 is a distributed systems module. Since most of you neither have multiple computers at home, nor a computer good enough to run multiple virtual machines, the distributed environment is simulated on a single machine. You are **not** allowed to use any multithreading in your program. If you do so, you will get zero for the entire project. In order to simulate the distributed environment you will have to create individual stand-alone agents/executables which each run on a **single** thread. Hence, if you have 8 cores on your machine, you can have 8 (or more) executables running independent and therefore utilizing 100% of your processor. However, you are not allowed to create a single executable that uses multiple threads to utilize 100% of your processor. If you want to fully utilize the processor, do not hardcode the number of cores, but rather detect them. Your code will be tested on various machines with a different core count, hence your code should automatically adapt to the system it is running on. Python has easy ways of detecting the system hardware. Remember that there is a difference between number of processors, number of cores, and number of threads.

5 Interface

All data retrieval and verification must be done through the provided program `rummy`. In order to facilitate communication between `rummy` and your program, you can execute `rummy` as an external command and then interpret the JSON results. Python and many other languages provide integrated support to execute external commands. `rummy` has a number of flags and options which are discussed below. Each result returned from `rummy` has the following JSON syntax:

{`"status" : string, "message" : string, "data" : string-or-list, "finished" : boolean`}

- `status`: Always present. Either `success` or `error`. Indicates whether or not your command executed successfully.
- `message`: Always present. A string that provides a human-readable description of the error or the successful command.
- `data`: Sometimes present. Returns a JSON string or list with the final data of your executed command.
- `finished`: Is only present if all maps and clues were solved and the treasure was found.

Note that each of the operations below require file read and write operations, encryption and decryption, and processing. Hence, you should try to reduce the number of commands you execute. For instance, if you want to remove three pirates from the crew, remove all three at once (faster) instead of one at a time (slower).

5.1 Test

Flag: `-wake` or `-w`
Parameters: None
Return: Standard message
Example: `>> rummy -wake`

This is a simple command to test if your interface is working. It doesn't do anything but print a statement.

5.2 Gather

Flag: `-gather` or `-g`
Parameters: None or the number of clues to generate
Return: Standard message
Example: `>> rummy -gather <number>`

This will instruct the Captain to collect all the clues he can find before looking for a crew. This takes the images from `data/maps` and preprocesses them into usable clues saved in `data/clues`. The command may take a very

long time to execute, depending on the number of images you selected and the number of processor cores available. This command only has to be executed once, and then you can reuse the clues for all your testing purposes. You can create your own or extend the current datatset. The parameter indicates the number of clues to generate. If no parameter is provided, it will use all 20 images. For testing purposes, use only one image, since this is faster and doesn't waste your time. However, the final project will be tested against all images. For your convenience, pregenerated clues can be found in `data/clues`.

5.3 Unlock

Flag: `-unlock` or `-u`
Parameters: None
Return: Standard message
Example: `>> rummy -unlock`

Since this is a distributed system, different processes might try to access files at the same time, causing a collision or inconsistency. For this reason files are locked allowing only one process to access them at a time. If you execute `rummy` in a normal way, all locks are released. However, if your program crashes or you exit it with `Ctrl-C`, some locks may not be released, causing `rummy` to wait indefinitely on the next execution. In such a case you can remove all locks by executing the unlock command.

5.4 Prepare

Flag: `-prepare` or `-p`
Parameters: None
Return: Standard message
Example: `>> rummy -prepare`

This will instruct the Captain to visit the pub in Port Noroyale to signup crew members. This command has to be executed only once before your computation commences and may take some time to execute.

5.5 Add

Flag: `-add` or `-a`
Parameters: None or the number of members to sign up
Return: List of new crew member IDs
Example: `>> rummy -add <number>`

This will instruct the Captain to add new pirates to the crew. The parameter indicates the number of pirates to add and defaults to one if omitted. A list of IDs for the new crew members are returned, which you will use at a later stage. Note that adding new crew members after being shipped out (refer to the `shipout` command), adds an additional time penalty to the execution. Hence, try to add all your crew members before shipping out. Note that a list is returned, even if only one member was added.

5.6 Remove

Flag: `-remove` or `-r`
Parameters: The IDs of the crew members to remove
Return: Standard message
Example:
```
>> rummy -remove "PirateID"
>> rummy -remove '["PirateID1", "PirateID2", "PirateID3"]'
```

This will instruct the Captain to remove pirates from the crew. Removing members does not introduce an extra time penalty, even when shipped out. You can either pass a single ID or a list of IDs. Remember to put the ID or the list of IDs in quotation marks.

5.7 Ship Out

Flag: `-shipout` or `-s`
Parameters: None
Return: Standard message
Example: `>> rummy -shipout`

This will instruct the Captain to ship out and start searching for the treasure. After this point your distributed processing starts and any new crew members added will induce and additional time penalty.

5.8 Clues

Flag: `-clues` or `-c`
Parameters: None or the ID of a pirate
Return: List of clues
Example:
```
>> rummy -clues
>> rummy -clues "PirateID"
```

This will instruct the Captain to hand out clues to individual crew members to go looking for the treasure. If no parameter is specified, a group of clues is returned. This list has objects, one for each pirate currently on the crew, containing the pirate's ID and a list of clues. Each clue has an ID and a string of data that must be processed later on. If you specify a pirate's ID for the command above, only a single clue object is returned, containing the clue ID and data for the specific pirate. Note that retrieving clues is time consuming and it is advised to retrieve all the clues at once (faster) instead of a single clue at a time (slower). Also note that although the individual clues are assigned to a specific pirate, any pirate can process the clues. The clues are simply linked to a pirate so that you can keep track of constant failures due to corrupt pirates (which you then might want to remove). Once a clue has been released with the above command, it will not be released again. Hence, you can call the command sequentially with a pirate's ID and a new clue is released on every iteration until none are left.

5.9 Verify

Flag: `-verify` or `-v`
Parameters: A single clue or a list of clues
Return: Standard message on success or list of failed clues
Example:
```
>> rummy -verify '{"id":"PirateID", "data":[{"id":"ClueID", "key":"ClueKey"}]}'
>> rummy -verify '[{"id":"PirateID1", "data":[{"id":"ClueID1", "key":"ClueKey1"},
   {"id":"ClueID2", "key":"ClueKey2"}]}, {"id":"PirateID2", "data":
   [{"id":"ClueID3", "key":"ClueKey3"}, {"id":"ClueID4", "key":"ClueKey4"}]}]'
```

This will instruct the Captain to verify if the pirate that solved the clues and checked the island for the treasure is talking the truth. You can pass in the clues from a single pirate or a list of clues from different pirates. Each clue contains the clue ID and the key generated by your algorithm. Again, any solved clue can be passed in under any pirate's ID. For instance, if a clue was released for pirate 1, pirate 2 may solve the clue, and pirate 3 may verify the key with the Captain. If all keys are successfully verified, a standard message is returned. Otherwise if some keys are incorrect, a list of failed clues is returned in the same format as the `clues` command. These clues have to be run through your algorithm again and the generated keys have to be reverified. This process continues until all keys for a given map are successfully verified. Once all keys are accepted, the next map is automatically unlocked and the new clues can be retrieved with the `clues` command and then verified with the `verify` command. Once all clues from all maps are verified, the treasure hunt is over. The final result will contain and additional attribute

`finished`. Again, verifying keys is time-consuming and it is better to verify all at once, instead of one key at a time.

6 Solving Clues

Each pirate or distributed agent has to solve clues. A clue is a string of characters that has to be scrambled in a certain way in order to get a key. This key is given to the captain for verification. Clues are solved through three main procedures, namely `dig in the sand`, `search the river`, and `crawl into the cave`. Each of these procedures require certain tools which are smaller operations, namely `shovel`, `rope`, `torch`, and `bucket`. These operations are discussed below. The input string is referred to as `clue`.

6.1 Shovel

Example Input: `40938FC0CB3F48B98C7546AD05CC7434`
Example Output: `00333444445567788899ABBCCCCCDFF0A2B3C`

1. Sort `clue` in ascending order. Digits should come before characters.
2. If the first character in `clue` is a digit, append the string "0A2B3C" to `clue`. Else if the first character is alphabetic, append the string "1B2C3D" to `clue`.
3. Remove the first character from `clue`.

6.2 Rope

Example Input: `40938FC0CB3F48B98C7546AD05CC7434`
Example Output: `A555BC25215CAB15B2ABA5Cd5B22AA5A`

For each character `x` in `clue`:

1. If `x` is a digit:
   (a) If `x` mod 3 is 0, change `x` to "5".
   (b) Else if `x` mod 3 is 1, change `x` to "A".
   (c) Else if `x` mod 3 is 2, change `x` to "B".
   (d) Else leave `x` as is.
2. Else if `x` is alphabetic:
   (a) Convert `x` to hexadecimal and subtract 11. Hence "A" becomes 0, "B" becomes 1, up to "F" which is 5.
   (b) If `x` mod 5 is 0, change `x` to "C".
   (c) Else if `x` mod 5 is 1, change `x` to "1".
   (d) Else if `x` mod 5 is 2, change `x` to "2".
   (e) Else leave `x` as is.

6.3 Torch

Example Input: `40938FC0CB3F48B98C7546AD05CC7434`
Example Output: `F9E8D701`

1. Set `x` to the sum of all digits in `clue`.
2. If `x` is less than 100, square `x`.
3. Convert `x` to a string.
4. If the string length of `x` is less than 10, remove the first character from `x` and prepend "F9E8D7" to it.
5. Else if the string length of `x` is greater equal to 10, remove the first 6 characters from `x` and append "A1B2C3" to it.

6.4 Bucket

Example Input: `40938FC0CB3F48B98C7546AD05CC7434`
Example Output: `80766FC0CB6F86B76C51084AD010CC5868`

For each character `x` in `clue`:

1. If `x` is a digit:
   (a) If `x` is greater than 5, subtract 2 from it.
   (b) Else if `x` is less equal to 5, multiply it by 2.
2. Else leave `x` as is.

6.5 Dig In The Sand

Example Input: `40938FC0CB3F48B98C7546AD05CC7434`
Example Output: `000000002222222...CCCCCCDFF0A2B3C`
Example Note: Due to the length of the output string, only the first and last parts are shown.

Dig in the sand for the treasure by using the shovel 100 times. Then use the bucket 200 times to get rid of the ground water. Then use the shovel another 100 times.

6.6 Search The River

Example Input: `40938FC0CB3F48B98C7546AD05CC7434`
Example Output: `604044FC0CB4F64B404C806068AD060CC80646`

Search the river for the treasure by using the bucket 200 times to empty the river.

6.7 Crawl Into The Cave

Example Input: `40938FC0CB3F48B98C7546AD05CC7434`
Example Output: `F9E8D7681`

Search the cave for the treasure by using the rope 200 times and then lighting up the torch 100 times.

6.8 Solve The Clue

Example Input: `40938FC0CB3F48B98C7546AD05CC7434`
Example Output: `6D4AD5F3CB5DD79A678D397CDB9AF434`

In order to solve the clue, the pirate has to first dig in the sand, then search the river, followed by crawling into the cave. The key is retrieved by calculating the MD5 hash of the result from the previous three actions and converting the hash to uppercase letters (digits stay the same).

7 Administration

This section provides some details on how to submit your project and how you will be evaluated.

7.1 Submission

You will have to submit your project by uploading it to the CS website no later than the **28th of May 2017**.

7.2 Code

All code must be able to compile and run on Linux. You can therefore not use programming languages or libraries that a specific to Windows or Mac. It is suggested that you use Python, while Java also provides cross-platform compatibility. If your code needs special instructions to compile and run, please provide them with your upload. To make things easier you could create a bash or Python script that automatically starts all your agents and begins the treasure-finding process.

7.3 Documentation

You are required to submit documentation in either PDF or Word format, no longer than **five** pages (excluding a cover page if you want to add one). In this document you have to explain the choices you made for each subsection under section 3 of this document. If you have special compilation and execution instructions, also add them to this document. You may add an additional page to the documentation for these instructions. If you need more than a page for the instructions, you are doing something wrong and you should talk to the lecturer.

7.4 Demonstration

You will be required to demonstrate your project and explain your implementation choices at the end of the semester between the 29th of May and the 2nd of June. Bookings will open at a later stage.

7.5 Mark Forfeiture

You will lose all the marks for this project for one of the following reasons:

- If you don't do or submit the project. Obviously.
- If you plagiarize or work in groups on this project. You may verbally share ideas and exchange you execution times, but your are not allowed to share code or implementation specifics.
- If you use multithreading in any way.
- If your code does not compile and run.
- If you do not show up for the project demonstration.

7.6 Bonus Marks

You will earn extra marks for any additional algorithm and functionality as discussed in section 3 of this document. In addition, bonus marks will be awarded to students who implement a faster and more efficient system than the lecturer. The lecturer's code is not fully optimized and it should not be difficult to beat it. Additionally, the student with the fastest system in the class will also earn extra marks, if and only if his/her code is faster than the lecturer's. The best student will be announced at the end of the semester.

7.7 Mark Allocation

The mark allocations for the project are as follows:

Description	**Weight**
Overall System	30
Execution Time	20
Agent Communication	20
Agent Leadership	10
Agent Corruption	10
Documentation	10
Total	**100**
Bonus - Faster Than Lecturer	5
Bonus - Fastest In Class	10
Bonus - Additional Features	10
Maximum	**125**