The Citadel System

Introduction to Operating Systems projects

An operating system is basically composed of 4 modules or subsystems:

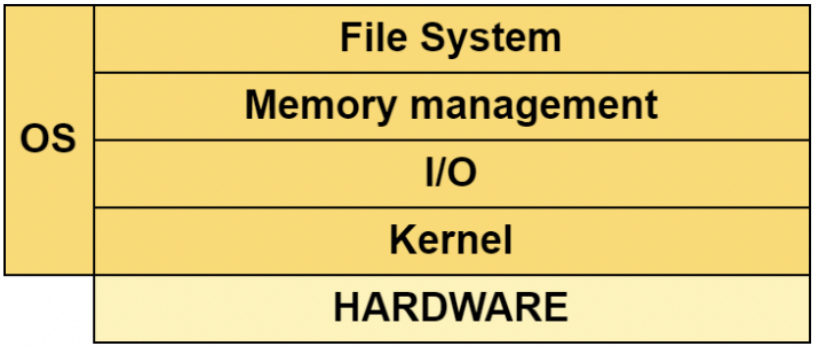

Figure 1. Architecture by layers of an Operating System.

KERNEL. It is the only layer that can inhibit interrupts. Its basic functions are representing running processes, controlling process concurrency, managing and handling interrupts, and providing the system with mechanisms for inter-process communication, synchronization, and mutual exclusion.

I/O SYSTEM. It is the only layer that can execute input/output instructions. Its function is communication with peripherals. Therefore, the upper layers, in order to interact with the hardware, must do so through this layer.

MEMORY MANAGEMENT. Together with the hardware, it enables the use of virtual memory. Furthermore, this layer is responsible (also jointly with the hardware) for ensuring data protection in memory and for enabling data sharing.

FILE SYSTEM. Provides the system with a structured view of the data stored on disk.

User applications are programs created by the user. When these programs require some functionality of the operating system, they make what is called a “system call,” which consists of calling functions provided by the different layers. That is, if the application wants to display a character on the screen, it must call a function in the I/O layer to do so.

The content of the Operating Systems course practice is oriented toward learning a distributed architecture through various communication mechanisms (mainly sockets), highlighting the challenges of process concurrency and multitasking. The main goal is to establish connectivity between different machines and enable the transmission of information among the various nodes of a distributed system. In addition, the use of kernel mechanisms of the operating system (shared memory, mutual exclusion methods, synchronization, and process creation) is required to complete the practical assignment.

NOTE 0. The four projects of the 2004-2005 academic year were named Atreides, Atreides++, Harkonnen, and Fremen. It was all in honor of Frank Herbert and his famous novel Dune. Many students remembered the origin and found out where the name came from.

NOTE 1. The four projects of the course 2005-2006 received the names Oedipus Rex, Sphinx, Antigone, and Teiresias. Like all SO projects, there is nothing random. The names came from the Oedipus tragedy from Sophocles and gave meaning to the project. The course began with a simple project 1, Oedipus the King. Oedipus lives happily as king while ignoring that he has killed his father and married his mother. Ignorance makes him happy. OS students live happily because project 1 is easy and they don't know what awaits them. Project 2 is the Sphinx. Oedipus must pass the Sphinx test, otherwise, it will kill him. Obviously, project 2 of the 2005-2006 year was the most difficult of all and if you did not pass it, you wouldn't pass the subject. Project 3 was called Antigone, who helps Oedipus once he falls into disgrace and despair. Antigone was the salvation of students who wanted to take the June exam and needed a long project delivered. I save you the details of Teiresias. Although many students followed the names and references, none of them fully grasped the meaning of the tragedy...

NOTE 2. The projects of the academic year 2006-2007 focused on Dante Alighieri's masterpiece, "Divina Commedia". The projects were called "Inferno", "Purgatorio" and "Paradiso". I guess it doesn't take much comment to understand what's going on in Project 1. The second project is softened but you need it if you want to take the exam. Finally, the third is relatively short and simple, a paradise with everything you have done before.

NOTE 3. The projects of the academic year 2007-2008 focused on the masterpiece of the Wachowski brothers, "Matrix", where humans (students) have to fight against machines to survive (pass the subject). The projects were called "Matrix", "Trainman" and "Keymaker". Matrix was the central server where customers, such as Nebuchadnezzar, had to connect in order to be able to exchange files and communicate through a distributed chat. They could also connect with other demons, such as Oracle or Link, who gave them advice. Trainman was a project of process load simulation and memory management. If you know the role of the Trainman in the film Matrix you will see that the relationship is direct. Finally, file encoding in EXT2. We considered that a good name to decode was the Matrix character called Keymaker.

NOTE 4. The projects of the academic year 2008-2009 focused on Isaac Asimov's masterpiece, "The Foundation". The projects were called: "Trantor", the central planet of the galaxy, therefore having the same magnitude as project 1 of Operating Systems; "Trevize, the loader", named after the director of the first foundation, with an unusual intuition, but with dangerous intentions; and "Pelorat, the file reader", name of the history professor who helped Trevize find planet Earth, the home to rest after passing the three operating systems projects.

NOTE 5. The projects of the academic year 2009-2010 focused on the Australian hard rock group AC/DC, as a tribute to the departure of the course's intern teacher for the last three years, Hugo Meza. The first project, under the name Highway to shell (a small modification of the name of the most popular song in the group), led the students through three phases. The first, Welcome to the Shell, welcomed the students to hell with the implementation of a shell command

OPERATING SYSTEMS Project 2021-2022 3 interpreter, under the name Malcolm (singer of the group). In the second phase, in order to escape the darkness in which they were, the students agreed with the devil (Dealing with the Devil), where they had to follow a protocol to communicate with the demon Angus (guitarist of the group). Finally, when they thought it was all over, they met Ballbreaker (an album they recorded in 1995), a name that does not require any explanation. The second project of 2009- 2010 was called Back in Black, the name of the album released in 1980, where students had to identify the format of a given volume and extract information from it.

NOTE 6. The projects of the 2010-11 academic year focused on the works of the controversial Donatien Alphonse François de Sade, better known as "Marquis de Sade". Thus, the project was titled "The 120 Days of Sodom". This was divided into three phases: "Le Château de Silling" was the first of them, where the students managed the behavior of the client of the application, which was named after the character in the novel "Blangis". Later the students entered the castle in a second phase called "Les quatre Madames", where Blangis connected to a demon called "Thérèse". Finally, the thing fainted in the third phase, where the students had to create the server "Libertinage", which, as the name suggests, is where the "festival" began.

NOTE 7. The projects of the 2011-12 academic year focused on the world-famous television series of the Simp(so)ns. This project was divided into five phases: "Homer's Shell", "Mou's Bar", "Clancy Wiggum", "Living in Springfield" and "Living in Springfield++", as a whole of these phases formed a system that allowed to execute commands locally, some of their own on a remote server and the activation of various services that showed mythical phrases of the series, all following a client-server architecture.

NOTE 8. The project of the 2012-13 academic year was called LsBox. The reason was quite simple: it was about designing and implementing a very similar (in fact simplified) system to the well-known Dropbox. The only curious detail was the project logo bearing an undercover Map of Australia as it was a small tribute to an OS intern that left these tasks.

NOTE 9. The project of the 2013-14 academic year was called LsHangIn. The reason was that it was a simplified version of the well-known Google Hangouts: it was about designing and implementing a multi-room chat system for users. In addition, each of the phases had a relation with the film The Hangover.

NOTE 10. The project of the 2014-15 academic year was called Gekko. Gekko is a businessman and the main protagonist of the film Wall Street, a direct relationship with the stock exchange and the goal of the project. But there were also several other tributes: TumblingDice, the fluctuation generator, is also a Rolling Stones song, and Dozer, the stockbroker, is one of the Matrix characters.

NOTE 11. The project of the 2015-16 academic year was called LsTransfer, the force awakens. It was a clear tribute to the return of the Star Wars saga. In addition, the server was called Naboo (the planet that appears in different episodes of Star Wars) and Gungan customers (inhabitants of this planet).

NOTE 12. The project of the 2016-17 academic year was called LsTinder, may the Love be with you. It was because of the clear similarity with the social network to which the project referred, and the motto another tribute to the Star Wars saga. In addition, the different processes were called Rick and Morty in reference to an American adult animated television series.

NOTE 13. The project of the 2017-18 academic year was called LsEat, may the Food be with you. First, the name is in reference to Just Eat fashion. The motto is another tribute to the Star Wars saga, in its imminent premiere of Episode VIII, The Last Jedi. In addition, the different processes had been named after Star Trek. Picard and Data characters and Enterprise, the ship.

NOTE 14. The project of the 2018-19 academic year was called the Cosgrove System, Stairway to heaven. First, Cosgrove is the last name of the family starring in the legendary Peter Jackson film Braindead, considered a classic of gore films of the time. The motto, Stairway to Heaven, was nothing related to the observatories of the project but a tribute to the Led Zeppelin song. In addition, the different processes had names of the protagonists of the Braindead film: McGruder, McTavish, Paquita, and Lionel.

NOTE 15. The project of the course 2019-20 called Cypher System was a tribute again to the Matrix, as it is being recorded for the 2022 Matrix IV, a myth in the world of science fiction. If you ran the cover script, it showed the green letters falling across the Matrix screen. Trinity is the main protagonist of Matrix, and the examples are always names of characters in the saga. There's nothing random.

NOTE 16. The 2020-21 project was called Overlook System. This was a tribute to Stanley Kubrick's legendary film The Shining, which celebrated its 40th anniversary in 2020. The Hotel was called Overlook and the image on the cover of the project was one of its mythical corridors. The different processes of the system design were Jack, Wendy, Danny, and Lloyd, which are the names of the personages of the saga. There is nothing random.

NOTE 17. The 2021-22 project was called the Arrakis System. This was a tribute to Denis Villeneuve's legendary 2021 film Dune but taking into account David Lynch's original from 1984. Arrakis is the planet where the majority of the film takes place. There were the Atreides (the good guys), the Harkonnen (the bad guys), and the Fremen (the native race of the planet Arrakis) who are important and key elements in the legendary film.

NOTE 18. The project of the academic year 2022-23 was called Eä System. This was a tribute to the series The Lord of the Rings: The Rings of Power. It is a television series based on the novel The Lord of the Rings and its appendices by J. R. R. Tolkien. It takes place thousands of years before Tolkien's The Hobbit and The Lord of the Rings in the Second Age of Middle earth. It is produced by Amazon Studios.

NOTE 19. The project for the academic year 2024-25 was named HAL 9000 System. This was a tribute to HAL 9000, mythical character from the well-known film series Space Odyssey; specifically, 2001: a Space Odyssey. 2001: A 1968 science fiction film directed by Stanley Kubrick, written by himself and Fleck C. Clarke. The film deals with topics such as human evolution, artificial intelligence, the future and extraterrestrial life. David Bowman and Frank Poole were

two of the protagonists and hence the names of the processes. The ship they were traveling on was called the Discovery (another process) and the Monolith was another important element in the film.

NOTE 20. The 2024–25 course project focused on the figure of Arthur Fleck, the Joker. The project was called *Mr. J. System*, in honor of how Harley refers to him. The system featured a central server, Gotham, and client processes named Fleck, which sent requests to the Enigma and Harley processes, all under the supervision of Arkham, a key element in the Batman universe.

From here, the interpretation of the names and concepts of this year's project (relating to the difficulty, content, and subject) is up to you... and we hope to see them in the final report!

The Citadel System

King Viserys is dead. The golden thread that once bound the realm has broken, and the trust among the Great Houses has evaporated. In this climate of tension, traditional trade routes are no longer safe, and alliances unravel as quickly as they are forged. Ravens fly toward castles bearing banners of dragons, green and black, but they do not carry messages of war, only an urgent need: to survive the coming political winter.

In this chaos, the Archmaesters of the Citadel have issued you a commission. Not to build a system of war, but a new network of trust and trade capable of operating in a fractured realm with no central power. You must design **The Citadel System**, a decentralized network that allows the Great Houses to forge alliances and then establish secure trade routes.

The main goal of the system is for the different Realms to be able to form **Alliances** among themselves in order to later **Trade**. To establish communication routes, the geography of the continent must be respected; therefore, a messenger must pass through all intermediate kingdoms (*hops*) carrying a raven until it reaches its destination. Once there, the sent raven will fly directly back to the origin with the reply.

To send multiple messages at once, the main process of each Realm (called the Maester) will rely on several assistants (Envoys). When a player wishes to initiate an action, such as requesting an alliance or trading, the Maester will assign the task to a free Envoy. That Envoy will handle all communication management for that specific task, allowing the Maester to manage multiple operations simultaneously. Coordination between the Maester and their Envoys will be one of the key design challenges.

Your task is to design and implement all the software needed to manage this ecosystem, following the specifications that will be provided. This project requires special emphasis on robustness, efficiency, and scalability, so that despite possible interruptions or sabotage attempts, the system can continue to operate smoothly.

To facilitate the process, development will be divided into incremental phases. Each phase will detail the functionalities to implement, the limitations, and provide tests to validate the progress of your work.

General Description

A first functional overview of the system's process architecture is shown in Figure 2. The architecture consists of a network made up of multiple independent **Maester** processes, where each process represents one of the Great Houses and communicates with the others to form a web of connections.

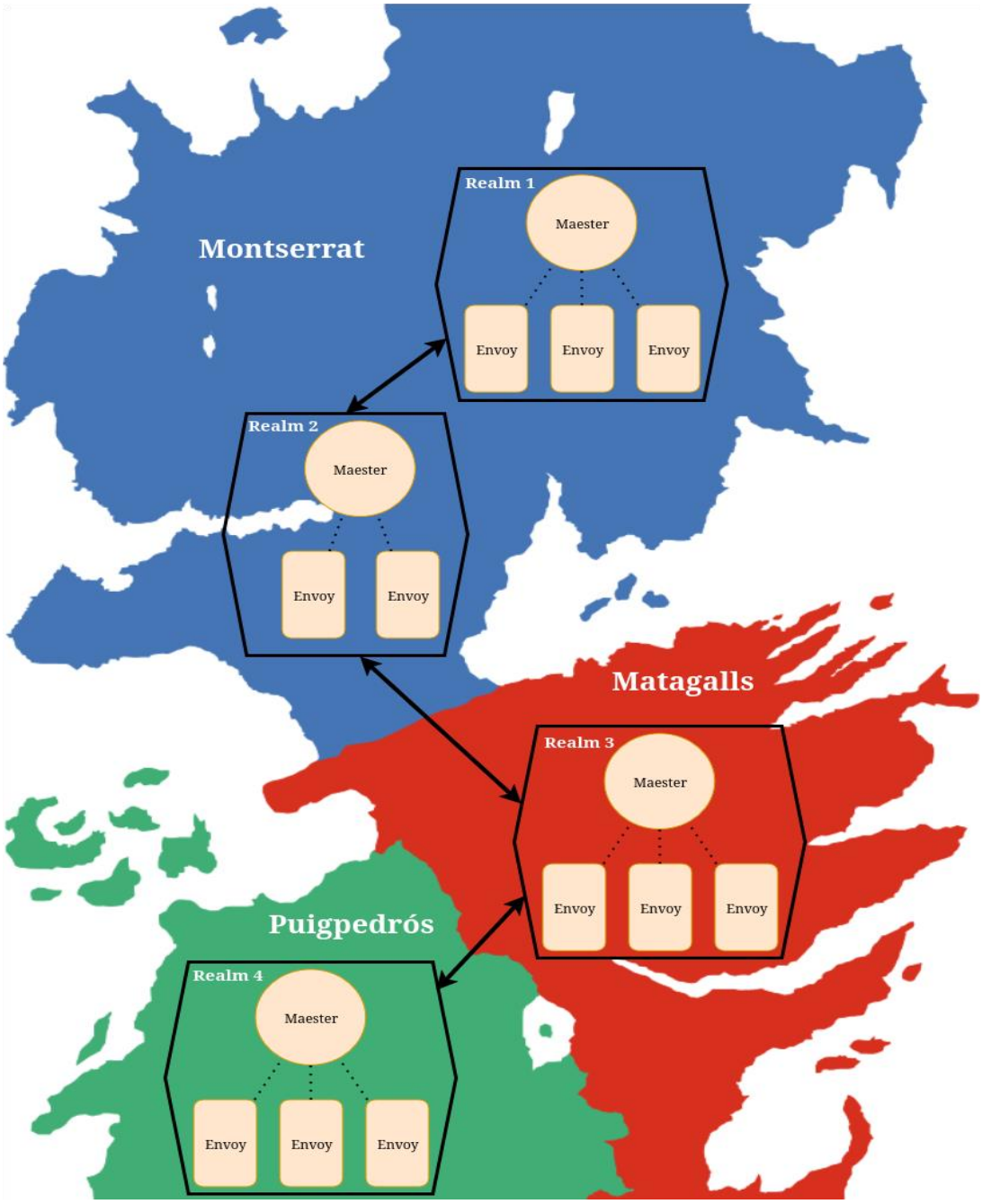

Figure 2. Citadel System: functional scheme.

The **Maester** processes act as the main nodes of the network, each representing a Noble House. The main goal is for them to establish **Alliances** among themselves and later be able to **Trade**. Communication will not be direct; a message must pass through several intermediate nodes (*hops*) to reach its destination, simulating the route of an emissary on horseback.

To handle concurrency and avoid bottlenecks in the main process, each **Maester** will have several assistants (**Envoys**). When a player wishes to start an action, such as requesting an alliance or sending a trade list, the Maester will assign the task to a free **Envoy**. That **Envoy** will manage all network communication for that specific task. This allows the **Maester** to handle multiple operations simultaneously, focusing on management and routing while the **Envoys** carry out missions.

In summary, the main objective of the system is to create a robust communication network where nodes can forge alliances and trade, managing the complexity of indirect communication and internal concurrency.

To design and implement the project, it is recommended to read the **full statement carefully.**

To make development easier, four phases have been planned:

1. **Phase 1**: creation of the Maester process. The network and Maester configuration file reading will be implemented, along with the command terminal containing all user commands to inspect realms, trade, and forge alliances.
2. **Phase 2**: implementation of basic connectivity among the different Maesters, including routing logic (*hops*). Alliances will begin to form, and trading will be enabled.
3. **Phase 3**: implementation of file transfer for alliances and trade. Each Maester will manage its product inventory and respond to purchase requests from its allies.
4. **Phase 4**: introduction of the **Envoy** processes. The code must be adapted so that the Maester delegates missions to the Envoys, allowing multiple alliance or trade requests to be sent concurrently.

Phase 1: Valar Compilis

In **Phase 1**, the design and programming of the system's main process, the **Maester**, will begin.

At this stage, the goal is to establish the foundations of the program: its initial configuration and user interface, without yet implementing network communication. The Maester process must be able to read its configuration files and respond to a set of user-entered commands (see Table 1).

Configuration file – Maester (*maester.dat*)

This text file will contain the basic information and routing table of the Maester in the following format.

- The name of the realm.
- The path to the folder where the user's files are located.
- The number of Envoys it will have.
- The IP and port (each on a separate line ending with '\n') where the Maester will listen for connections.
- A section beginning with --- ROUTES --- that defines the connections. Some entries will have a known IP and port, while others may contain *.*.*.* to indicate that the realm exists but its route is not directly known.

An example of a configuration text file is shown in Figure 3.

```
Dragonstone
/dragonmont
3
192.168.1.3
9003
--- ROUTES ---
DEFAULT 192.168.1.5 9002
KingsLanding 192.168.1.5 9002
TheVale 192.168.1.3 9005
Driftmark *.*.*.* 0
```

Figure 3. Configuration file of a Maester process.

Inventory File – Maester (*stock.db*)

Each Maester will manage their realm's inventory through the **official ledger.** This ledger is not a simple scroll; it is written in a secret code that only the Maesters can decipher, protecting the house's trade secrets.

Technically, it is a **binary file**. Each entry in the ledger defines a product with the following fields:

- **Name:** char name[100];
- **Quantity:** int amount;
- **Weight:** float weight;

The inventory record must be preserved between executions. Any change in stock must be reflected in the inventory file to ensure persistence.

Command Terminal

The Maester process must handle an interactive terminal. Most of the time, it will display activity logs of the requests it receives or forwards to the next node. The terminal must always remain active so that the user can enter custom commands. These commands are *case-insensitive.*

COMMAND	DESCRIPTION
`LIST REALMS`	Displays all realms listed in the configuration file.
`PLEDGE <REALM> <sigil.jpg>`	Initiates an alliance request to another realm.
`PLEDGE RESPOND <REALM> ACCEPT / REJECT`	Responds affirmatively or negatively to an alliance request you have received.
`LIST PRODUCTS <REALM>`	Requests the list of products from a realm (requires a prior alliance).
`LIST PRODUCTS`	If no realm is specified, it will display your own products. No alliance is required, since they belong to your own Maester.
`START TRADE <REALM>`	Starts an interactive session to create a "shopping list." Displays the available products (previously obtained with LIST PRODUCTS) and allows the user to add them to the list.
`PLEDGE STATUS`	Shows the status of your alliances (pending, allied).
`ENVOY STATUS`	Shows the status of all your Envoys.
`EXIT`	Ends execution.

Table 1. List of extended commands.

For the START TRADE command, a menu must be displayed like the one shown in the example, allowing the user to build a "shopping list" to send to the requested realm.

For this first phase, the commands in this menu must be detected using invented products (you may use your own). When a trade session ends, the shopping list must be written to a file. This file will later be used to send data to other realms. The format of this file is flexible, but it must be a text file.

```
$montserrat:> Maester config.dat stock.db

Maester of Dragonstone initialized. The board is set.
$ PLEDGE TheVale sigil.jpg
Pledge sent to TheVale. Envoy 1 is on its way.
$ PLEDGE STATUS
- TheVale: PENDING
- KingsLanding: PENDING
- Driftmark: PENDING
$ ENVOY STATUS
- Envoy 1: ON_MISSION (PLEDGE to TheVale)
- Envoy 2: FREE
- Envoy 3: FREE
$ LIST PRODUCTS KingsLanding
ERROR: You must have an alliance with KingsLanding to trade.
$ PLEDGE KingsLanding sigil.jpg
Pledge sent to KingsLanding. Envoy 2 is on its way.
$ something else
Unknown command
$ ENVOY STATUS
- Envoy 1: ON_MISSION (PLEDGE to TheVale)
- Envoy 2: ON_MISSION (PLEDGE to KingsLanding)
- Envoy 3: FREE
$
>>> Alliance with The Vale forged successfully!
$ PLEDGE STATUS
- TheVale: ALLIED
- KingsLanding: ON_PENDING
- Driftmark: PENDING
$ LIST PRODUCTS TheVale
Listing products from TheVale:
1. Myrish Lace (150 units)
2. Sweetwine (80 units)
$ START TRADE TheVale
Entering trade mode with TheVale.
Available products: Myrish Lace, Sweetwine.
(trade)> add Myrish Lace 10
(trade)> add Sweetwine 5
(trade)> send
Trade list sent to TheVale.
… next figure
```

Figure 4.1. Execution of a Maester process.

```
Trade list sent to TheVale.
$ PLEDGE Driftmark sigil.jpg
Pledge sent to Driftmark. An envoy is on its way.
$
>>> Pledge to King's Landing has failed (TIMEOUT).
$ PLEDGE STATUS
- TheVale: ALLIED
- KingsLanding: FAILED
- Driftmark: PENDING
$ LIST PRODUCTS
--- Trade Ledger ---
Item                    | Value (Gold) | Weight (Stone)
-----------------------------------------------------------
Antlered Helm           | 250          | 4.5
Warhammer of Bronze     | 400          | 8.0
Stormlands Ale Barrel   | 80           | 10.0
Ship Timber Plank       | 60           | 15.0
Salted Fish Barrel      | 55           | 20.0
Wine Cask               | 120          | 5.0
Leather Riding Boots    | 90           | 2.0
Bronze Helm             | 140          | 3.5
Woolen Blanket          | 30           | 2.5
Iron Dagger             | 35           | 1.2
Grain Sack              | 40           | 10.0
-----------------------------------------------------------
Total Entries: 11

$ EXIT
The Maester of Dragonstone signs off. The ravens rest.

$montserrat:>
```

Figure 4.2. Execution of a Maester process, command examples.

Requirements:

- Design and program the maester executable.
- On startup, the Maester must process its configuration files (.dat and .db) and store the information in the corresponding data structures.
- The Maester must recognize all commands listed in Table 1. For this phase:
    - Fully implement the local functionality of the following commands: LIST REALMS, LIST PRODUCTS (without realm), EXIT, and START TRADE.
    - For the remaining commands, the program should simply display "Command OK" to indicate that the command was recognized.
    - The command recognition system must be robust. If a command is invalid, it should display "Unknown command". If a command is valid but incomplete (e.g., PLEDGE without arguments), it should display a free-form help message such as: Did you mean to send a pledge? Please review syntax.

Phase 1 Considerations:

- Once all processes have finished execution, any significant dynamically allocated memory must be **freed**.
- If realm names contain the character &, it must be removed. This is due to the system's communication protocol described in the Annex. The protocol is *case-sensitive*.
- Commands are *case-insensitive*.
- You may assume that the **configuration file format is correct**.
- The use of functions such as *printf, scanf, gets, puts,* etc. is **forbidden**. Interaction with the screen and files must be done exclusively through *read* (for reading) and *write* (for writing). The use of *asprintf* and similar functions is allowed.
- The use of *system, popen, stat*, or any related variants is **forbidden**.
- The application must remain stable and function correctly. Infinite loops, *core dumps*, busy waits, or compiler *warnings* are not acceptable. All possible error conditions must be properly handled; if an error occurs, the program **must inform the user** and continue running whenever possible.
- From now until the end of the project, the Maester process may be terminated either through the appropriate command or by pressing CTRL+C. The system must handle this **gracefully**, releasing all resources and closing everything properly.
- A *Makefile* is mandatory for generating the executable.
- It is not enough for the file you upload to have a .tar extension; it must be extractable using the tar command. Any project or *checkpoint* that cannot be extracted this way **will not be graded**.
- The code must be properly modularized. It is not permitted to place everything in a single .c file, and a *Makefile* is mandatory. If the submission does not compile with the *make* command for any reason, it will be graded as not approved (2).
- Alongside the code, you must submit an explanation of how the phase was designed and structured. **It is mandatory to perform this analysis before programming and validate it with the course instructors**. **See Annex III for more details.** The report must clearly and understandably include:
    - Diagrams explaining the created processes.
    - The data structures used and the justification for their selection.

Phase 2: Kingsroad

In this second phase it will be necessary to implement the connections between the different realms of the system. The Maester processes will now start communicating between them to forge alliances or, afterwards, trade.

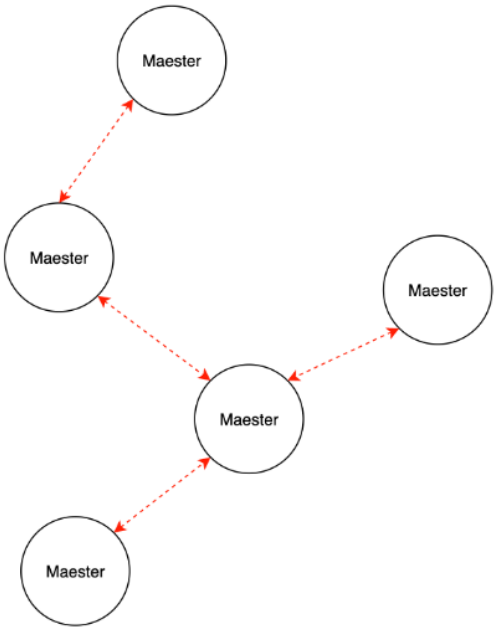

Figure 5. Functional structure to be designed and implemented.

Every Maester will act as a node in a distributed network, handling both its own communication as well as acting as an intermediate node (hop) for communication from neighboring realms. Since the different Maester processes can be in different physical machines, the connection between them will need to be implemented with **sockets**, and following the **communication protocol** located in the Annex.

By default, the realms start with no alliances established, but the user will be able to create them from every Measter process. To establish an alliance, the Maesters send a sigil (an image) that certifies the message is official. The process to follow is:

1. One of the Maesters, Maester “A” initiates an alliance petition with realm “B” sending the first message (TYPE 0x01). If “A” does not have “B” in its routes table, it will need to send the message through as many hops as necessary.
2. Afterwards, upon receiving the first message, Maester “B” confirms it’s ready to receive the sigil (TYPE 0x31).
3. Maester “A” sends the whole sigil (TYPE 0x02).
4. Finally, Maester “B” decides if it accepts or rejects the alliance petition (TYPE 0x03).

However, in **phase 2** the alliance forging process will be simplified, and will not implement the sigil image data sending, only the *header*. Which means that, when an alliance is attempted to be created between two realms, the first Maester will send the alliance petition to the second, and the second will answer with the confirmation/rejection of the alliance, but at no point will the sigil image be sent (TYPE 0x31 and 0x02 messages). This will be implemented in **phase 3**.

System connectivity and message routing (hops)

For all the communication network to function properly, when a Measter receives a message, it must determine if it is the final destination or if it needs to act as an intermediate node (hop) and forwards it to another realm. To do so, the Maester process will follow this process:

1. When the DESTINATION field is my realm:
    a. Check that the message is correct (using the *checksum* value).
        i. If the check is incorrect, send a NACK (TYPE 0x69) to the node that sent me the message (previous hop) and discard the received incorrect message.
    b. If it is correct, send an ACK (type depends on the type of message) to the sender.
    c. Process the message.
2. If the DESTINATION field is not my realm:
    a. Check that the message is correct (using the *checksum* value).
        i. If the check is incorrect, send a NACK (TYPE 0x69) to the sender of the message and discard the received incorrect message.
    b. Look up in the routes table who it is destined for.
    c. If the route exists, forward it to the IP:port in the routes table.
    d. If it does not exist, forward it to the DEFAULT route.
    e. If there is no DEFAULT route, show an error and discard the message (the origin node, the sender, will detect the error by *timeout*).

Take into account that, at the start of the system, all communications will follow this mechanism. To exemplify, if realm “A” has to send a message to “C”, going through realm “B”, all communications to establish an alliance between “A” and “C” will need to go through realm “B” (except for the last ACK, which goes back directly from “C” to “A”). Nevertheless, when the alliance is established between realms “A” and “C”, they will be able to communicate directly (without going through “B”), using the ravens and optimizing trade.

Trading with allies

Once an alliance is established with another realm, trade is available between them (otherwise, all trade messages will be rejected). Trade is based on requesting product lists and exchanging goods. However, in this **phase 2** we will simplify this operation, and all trade messages will return a valid ACK, without processing the message internally nor updating the stock. This will be implemented in **phase 3**.

Finally, when a Maester process decides to end its execution, it must alert all the realms it's allied with (TYPE 0x27).

```
$montserrat:> Maester dragonstone.dat stock.db

Maester of Dragonstone initialized. The board is set.
$ PLEDGE TheVale sigil.jpg
Pledge sent to TheVale. Envoy 1 is on its way.
>>> Alliance with TheVale forged successfully!
$ LIST PRODUCTS TheVale
Direct route available (allied). No hops required.
List products: OK (Pending F3)
$ EXIT
The Maester of Dragonstone signs off. The ravens rest.

$montserrat:>
```

```
$montserrat:> Maester kingslanding.dat stock.db

Maester of KingsLanding initialized. The board is set.

>>> Received hop: Dragonstone → TheVale
Found route: TheVale → 192.168.1.4:9005
Forwarding PLEDGE request...
```

```
$matagalls:> Maester vale.dat stock.db

Maester of TheVale initialized. The board is set.
>>> Alliance request received from Dragonstone.
$ PLEDGE RESPOND Dragonstone ACCEPT
Alliance with Dragonstone established.
>>> Trade request received from Dragonstone (direct).
Order processed successfully. Stock updated.
```

Figure 6. Execution example of 3 Maesters ("A" an "B" on montserrat; "C" on matagalls); where "A" forges an alliance with "C", going through "B".

Phase 2 Considerations:

- Those of Phase 1 are still valid.
- In this Phase, image (sigils) transfers and trade orders should NOT be implemented.
- It will be necessary to manage the crashes of all processes and maintain the stability of the system (sending error messages when things don't work).
- Along with the code, an explanation of how the phase has been designed and structured must be provided. It is highly recommended that, before starting to program, you carry out this analysis and validate it with the interns of the subject. **See Appendix IV for more details**. The report must include in a clear and understandable manner:
    - Diagrams explaining processes have been created, different communications between processes, etc.
    - Data structures used and their justification.
    - System resources used (*threads*, *forks*, *signals*, *sockets*, semaphores, *pipes*, etc.) with their justification.
    - Optionals implemented.

Phase 3: Dance of the sigils

In this phase, one of the most awaited functionalities of the Citadel System will be deployed: file transfers between realms. After having established basic connectivity and the first alliances in the previous phase, now is the moment to start moving the **sigils** and **commercial files** (product lists and orders) that support the diplomatic and economic relationships between the Great Houses.

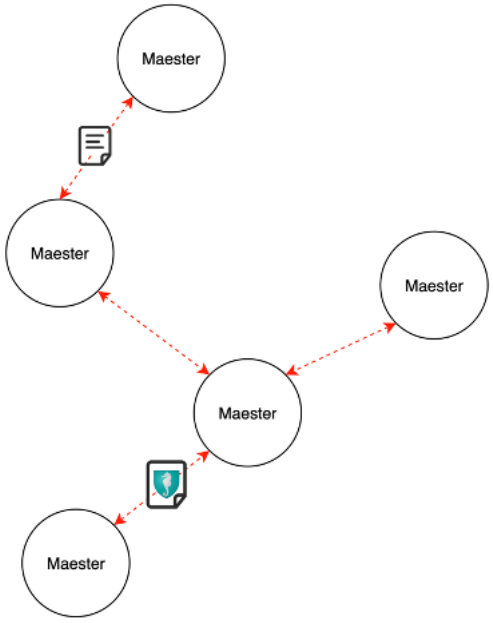

Figure 7. Functional diagram of the sending of sigils and trade

Every Maester needs to be capable of not only sending and receiving these files, but also verifying their integrity. For this purpose, the md5sum command from the operating system will be used, which allows verifying the received contents match exactly with the original. Only if the verification is correct will the alliance petition or trade order be considered valid.

The transfer process is formed of two different parts:

- Start of the transfer (header): the essential file information is sent (name, size, MD5SUM value).
- Data transfer: the file will be sent fragmented into data frames.

The sending of the header will have been done in the previous phase, now it's time to expand that functionality to also send the file data. Once the transfer is completed, the receiving realm must respond with an acknowledgement message, confirming the received file is valid or if it has been corrupted.

The functionality of Phase 3 covers two main scenarios:

- Alliances: transfer of the sigil.jpg file, symbolizing the pact between two houses.
- Trade: transfer of the product lists and orders as files.

The design decisions of the communication will need to be explained accordingly in the practice report, adding a diagram with the implemented communication.

Moreover, this phase also introduces stock management in all realms. Every Maester must be capable of accessing their ledger (stock.db) and respond appropriately to allies' requests. If a realm receives an order and doesn't have enough stock, it will need to reject the order. If the product doesn't exist, it will also need to indicate it with the appropriate frame.

Since the communications in the Citadel System is done through hops, all of these files will traverse intermediate realms, which will not modify neither the ORIGIN field nor the DESTINATION field; they will just forward the frame. If there is no known route, the intermediate node must respond with an error frame. If the realms are already allied, they will be able to communicate directly, given that when allying they have shared their IP + Port.

In this phase, we will see for the first time how the Maesters become real, reliable communication nodes: able to exchange binary data, validate it, and update their local inventories accordingly.

$montserrat:> **Maester dragonstone.dat stock.db** Maester of Dragonstone initialized. The board is set. **$ PLEDGE TheVale sigil.jpg** Pledge sent to TheVale. Envoy 1 is on its way. >>> Alliance with TheVale forged successfully! **$ START TRADE TheVale** Direct route available (allied). No hops required. Entering trade mode with TheVale. No products available. Use LIST PRODUCTS first. (trade)> **exit** **$ LIST PRODUCTS TheVale** *… continue on next page*	$puigpedros:> **Maester kingslanding.dat stock.db** Maester of KingsLanding initialized. The board is set. Received hop: Dragonstone → TheVale (PLEDGE) Found route: TheVale → 192.168.1.4:9005 Forwarding...

```
Listing products from TheVale:
1. Myrish Lace (150 units)
2. Sweetwine (80 units)

$ START TRADE TheVale
Direct route available (allied). No hops
required.
Entering trade mode with TheVale.
Available   products:   Myrish   Lace,
Sweetwine.
(trade)> add Myrish Lace 10
(trade)> add Sweetwine 5
(trade)> send
Trade list sent to TheVale.
>>> Order accepted by TheVale.

$ EXIT
The Maester of Dragonstone signs off.
The ravens rest.

$dragonstone>
```

```
$matagalls:> Maester vale.dat stock.db

Maester of TheVale initialized. The board is set.
>>> Alliance request received from Dragonstone.
$ PLEDGE RESPOND Dragonstone ACCEPT
Alliance with Dragonstone established.

>>> LIST PRODUCTS request from Dragonstone.
Sending product list.
Products delivered.

>>> Trade request received from Dragonstone.
Order processed successfully. Stock updated.
```

Figure 8. Execution example: 3 Maesters ("A" on montserrat, "B" on puigpedros; "C" on matagalls). "A" trades with "C", and goes through "B" only to establish the alliance.

Phase 3 considerations:

- Those of the previous phases are still valid.
- The alliance (sigils) and trade (lists and orders) files must be transferred in full and verified with **md5sum**.
- It is not allowed to use programmed code to make MD5SUM. You need to run the md5sum command that provides the same *bash* as the operating system (*md5sum*).
- The realms must update the local inventory according to the accepted orders, ensuring the changes are persistently stored in *stock.db*.
- Along with the code, an explanation of how the phase has been designed and structured must be provided. It is highly recommended that, before starting to program, you carry out this analysis and validate it with the interns of the subject. **See Appendix IV for more details**. The report must include in a clear and understandable manner:
    - Diagrams explaining processes have been created, different communications between processes, etc.
    - Data structures used and their justification.
    - System resources used (*threads*, *forks*, *signals*, *sockets*, semaphores, *pipes*, etc.) with their justification.
    - Optionals implemented.

Phase 4: The Council of Envoys

In the final phase of the practice, the realm's range of actions is expanded. Until now, a Maester already managed multiple tasks in parallel: their main function was to constantly receive the incoming ravens, either to forward them to the next hop along the route, acting like a router, or to process them if they were intended for their own realm. In addition to this continuous task of reception and routing, the Maester could only initiate and manage a single mission of their own at a time, whether it was an alliance or trade request.

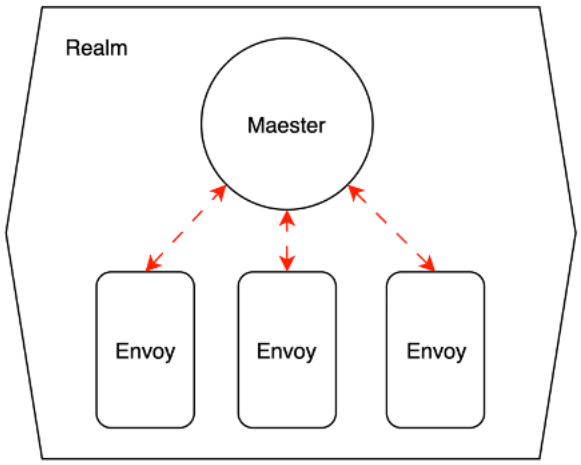

Figure 9. Realm processes functional diagram

In this phase, we will implement the Council of Envoys to remove this limitation. The number of Envoys is specified in the configuration file. When the user wishes to start a mission, the Maester will look for a free Envoy and assign the task to it. That Envoy will handle all aspects of communication regarding that task, freeing the Maester to continue performing its management and routing duties.

Considerations Phase 4:

- Those of the previous Phases are still valid.
- Sockets nor secondary memory cannot be used for the Envoy <-> Maester communication.
- Mutual exclusion must be guaranteed for all shared resources.

Delivery and planning requirements

This practice will have different **control points or *checkpoints*** to be able to carefully monitor it and guarantee the consolidation of each of the phases incrementally. Specifically, the following calendar of deadlines will be followed:

CONCEPT	DEADLINE
Phase 1	26/10/2025
Phase 2	23/11/2025
Phase 3	14/12/2025
Final Delivery 1 (out of 10)	06/01/2026
Final Delivery 2 (out of 10)	01/02/2026
Final Delivery 3 (out of 8)	17/05/2026
Final Delivery 4 (out of 6)	14/06/2026

Checkpoints are not mandatory, although they are highly recommended to guarantee the robustness of the practice and to know that you are on time to be able to deliver at the end of the semester. These *checkpoints* weigh 10% of the project grade.

It is also mandatory **to validate the overall design of the practice** with the interns before starting phase 1. This way you can guarantee that the implementation will not suffer from inconsistencies that cannot be fixed by later phases. For this reason, it is necessary to take advantage of the hours of doubts and **bring the printed designs** to be validated. It should be remembered that a working practice is not a guarantee that it will be valid, as the entire design may not meet the requirements of the statement. That is why it is very important to have guarantees about the design to be implemented. You can review Annex IV for design guidelines.

The deliveries will be made in the eStudy with a file that includes the source code (.c, .h files and the makefile) of the practice, working completely on Montserrat, as well as the data and configuration files used. The file must be in .tar format. You can generate it with the following command:

```
tar cf Gx_Fn.tar *.c *.h makefile *.ext_files
```

where Fn is the Phase number to be delivered. For example, group 12 would send G12_F1.tar for the delivery of the *Phase 1* checkpoint.

For the final submissions (from January onward), n = 4, corresponding to Phase 4.

Any submission that does not follow the submission format will not be corrected.

In the partial deliveries, it is also necessary to include the report that is being carried out so that its evolution can be assessed. From Phase 2 onwards, it must contain the design of the practice.

In the final deliveries it will also be necessary to deposit the **final report in PDF format** in the tar. The report must consist of the points indicated below in Annex I.

Annex I: Report Contents

The report must be properly formatted documentation. It must contain the following sections:

1. **Cover**
2. **Table of Contents**
3. **Design**: an explanation of how the project has been designed and structured. It can be explained phase-by-phase or the project as a whole. It must include clearly and understandably:
   a. Diagrams explaining what processes have been created, the different communications between processes, etc.
   b. Data structures used and their justification.
   c. System resources used (signals, sockets, semaphores, pipes, etc.) with their justification.
   Optional phases implemented.
4. **Observed problems and how they have been solved.**
5. **Temporal estimation:** time spent on the whole development of the project for each of the students. The categories to consider are:
   a. Research: time spent looking for tools and/or learning components to implement (without counting the time spent on the lab sessions).
   b. Design: time spent designing the project.
   c. Implementation: coding time.
   d. Testing: time spent testing.
   e. Documentation: time spent writing the report and internal documentation of the code (comments, etc.).
6. **Conclusions and proposals for improvement.**
7. **(Optional) Explanation of ALL the names of the project processes. What theme did we base this year on to make the statement?**
8. **Bibliography used** (in a correct bibliographic format IEEE, APA, etc.): both bibliographic resources and links.

The writing and the grammatical and orthographic correctness will be assessed. The report must have **numbered pages**.

The report must be delivered in **PDF format** in the final delivery, but it is recommended to elaborate it during the implementation of the project.

Annex II: general communication protocol

To carry out communication between processes on different machines, a specific frame sending protocol will be used, which will be explained below. It should be noted that messages will be sent via *sockets* using connection-oriented communication.

In this protocol, a **single type** of frame will be used. These are of fixed dimension **(320 Bytes)** and always formed by the following 6 fields:

Type: Describes the type of the frame in hexadecimal format 1 character that will contain a frame ID.

Origin: A 20-byte field with the IP + Port of the realm sending the frame.

Destination: 20-byte field with the name of the destination realm.

Data Length: A 2-byte field in numerical format that is used to indicate the length of the data field.

Data: This field is used to store values or data that the frame must send. The length of this field directly depends on the total length of the frame and the value of data_length.

Checksum: A 2-byte field that will help validate that the frame has not had any problems. This field will be calculated by adding all the bytes of the frame (not counting the checksum itself) and doing module 2^16. It will be important to fill in each sent frame with this checksum and validate it when it arrives at the respective servers.

TYPE (1 Byte)	ORIGIN (20 Bytes)	DESTINATION (20 Bytes)	DATA_LENGTH (2 Bytes)	DATA (320 - 45 - X Bytes)	CHECKSUM (2 Bytes)

It is very important in order to guarantee good communication with the process that the definition of the structure of the communication frames is strict, in order and format, as described above. This means that any change in the type of the fields or their order within the structure to be designed will cause the frames received or sent to the process to not comply with the communication protocol and, therefore, this communication will not work correctly. If the size of the data to be sent is less than 320, the frame must be filled in with the *corresponding padding.*

It is completely normative **to send entire serialized frames** in a single write to the *socket*. That is, read with a single read, and write with a single write of 320B.

For **all** frames, the CHECKSUM and TIMESTAMP fields will be calculated in the same way:

CHECKSUM: Sum of all bytes in the frame % (2^16). You can use the value 65536 directly. (Sum of bytes % 65536).

For simplicity, they have not been included in the individual explanations of each frame since it would become repetitive, but they must be in all of them. Everything in between <> must be replaced with the corresponding value. **The <> are NOT part of the frame.**

ALLIANCE REQUEST (HEADER) - MAESTER

Maester A → Maester B

When a Maester wants to initiate an alliance, they first send the general information and the data for the sigil file.

- TYPE: 0x01
- ORIGIN: <IP:Port of the initiating realm (not the hop)>
- DESTINATION: <Name of the destination realm>
- DATA_LENGTH: Data length.
- DATA: <RealmName>&<SigilName>&<FileSize>&<MD5SUM>

Where:

- RealmName: Name of the realm initiating the request.
- SigilName: Name of the sigil file.
- FileSize: in bytes, expressed in ASCII format.
- MD5SUM: the 32 characters of the MD5SUM.

After sending a frame of type 0x01, the **ACK FILE** (0x31) must be sent**.**

NOTE: This is the frame used in Phase 2. Starting from Phase 3, the file data itself must also be sent afterwards with the frame:

SIGIL TRANSFER (DATA) - MAESTER

Maester A -> Maester B

- TYPE: 0x02
- ORIGIN: <IP:Port of the initiating realm (not the hop)>
- DESTINATION: <Name of the destination realm>
- DATA_LENGTH: Data length.
- DATA: *file_data.*

After sending a 0x02 frame, an **ACK MD5SUM** (0x32) frame must always be sent.

ALLIANCE REQUEST RESPONSE - MAESTER

Maester B -> Maester A

When a Measter receives an alliance request and the md5sum is OK, it will reply with an acceptance or a rejection. This will have the following format:

- TYPE: 0x03
- ORIGIN: <IP:Port of the origin realm, the one replying to the alliance petition>
- DESTINATION: <Name of the realm requesting an alliance>
- DATA_LENGTH: Data length.
- DATA: <ACCEPT/REJECT>

It is important that, in this step, the origin IP is changed to that of Maester B, who is accepting (or rejecting) the alliance. Up to this point, during intermediate hops, the original IP had to be preserved so the destination realm could reply directly to the origin realm. In this reply, however, the origin realm must be able to learn the IP + Port of its new ally.

PRODUCT LIST REQUEST - MAESTER

Maester A -> Maester B

When a Maester wants to check the available products in an allied realm, it sends an initial request.

- TYPE: 0x11
- ORIGIN: <IP:Port of the initiating realm (not the hop)>
- DESTINATION: <Name of the destination realm>
- DATA_LENGTH: Data length.
- DATA: <RealmName>

Where:

- RealmName: the name of the realm initiating the request.

PRODUCT LIST RESPONSE (HEADER) – MAESTER

Maester B → Maester A

The allied realm prepares a file containing its inventory list and sends the header frame:

- TYPE: 0x12
- ORIGIN: <IP:Port of the allied realm (the one responding)>
- DESTINATION: <Name of the realm that requested the product list>
- DATA_LENGTH: Data length.
- DATA: <FileName>&<FileSize>&<MD5SUM>

Where:

- FileName: can be “products.db” or an equivalent name.
- FileSize: size in bytes, ASCII.
- MD5SUM: 32 characters ASCII.

After sending the type 0x12 frame, the **ACK FITXER** (0x31) frame must be sent.

PRODUCT LIST TRANSFER (DATA) – MAESTER

Maester B → Maester A

After the header frame, the allied Maester sends the inventory list file in binary data blocks.

- TYPE: 0x13

- ORIGIN: <IP:Port of the allied realm (the one sending the inventory)>
- DESTINATION: <Name of the realm that requested the product list>
- DATA_LENGTH: Data length.
- DATA: *file_data.*

After sending all 0x13 frames, the **ACK MD5SUM** (0x32) frame must be received.

ORDER REQUEST (SHOPPING LIST HEADER) – MAESTER

Maester A → Maester B

When a Maester wants to send an order, it first transmits the information of the shopping list file:

- TYPE: 0x14
- ORIGIN: <IP:Port of the origin realm>
- DESTINATION: <Name of the ally realm>
- DATA_LENGTH: Data length.
- DATA: <FileName>&<FileSize>&<MD5SUM>

After sending a frame of type 0x14, the **ACK FITXER** (0x31) frame must be received.

ORDER TRANSFER (DATA) – MAESTER

Maester A → Maester B

After the header frame, the origin Maester sends the file containing the list of desired products.

- TYPE: 0x15
- ORIGIN: <IP:Port of the origin realm>
- DESTINATION: <Name of the ally realm>
- DATA_LENGTH: Data length.
- DATA: *file_data.*

After sending all type 0x15 frames, the **ACK MD5SUM** (0x32) frame must be received.

ORDER RESPONSE – MAESTER

Maester B → Maester A

When the allied realm processes the received order, it replies with the result:

- TYPE: 0x16
- ORIGIN: <IP:Port of the realm responding to the request>
- DESTINATION: <Name of the realm that sent the order request>
- DATA_LENGTH: Data length.
- DATA: OK **or** REJECT&<reason>

Reason example:

- OUT_OF_STOCK if there is not enough quanitity of a product.
- UNKNOWN_PRODUCT if the product does not exist in the catalog.

MAESTER DISCONNECTION – MAESTER A → MAESTER B (ally)

When a Maester finishes execution, it must notify (of its disconnetion) all realms with which at has an active alliance.

- TYPE: 0x27
- ORIGIN: <IP:Port of the disconnecting realm>
- DESTINATION: <Name of the ally realm>
- DATA_LENGTH: Data length.
- DATA: DISCONNECT

Expected behaviour:

- The allied realms mark the alliance as inactive.
- No acknowledgement (ACK) is required: this is a notifiaction only.

ERROR: UNKNOWN REALM – INTERMEDIATE NODE → ORIGIN

When there is no specific route nor a DEFAULT route to the DESTINATION.

- TYPE: 0x21
- ORIGIN: <IP:Port of the intermediate node generating the error>
- DESTINATION: <Name of the realm initiating the petition>
- DATA_LENGTH: Data length.
- DATA: UNKNOWN_REALM

ERROR: UNAUTHORIZED / NO ALLIANCE – RECEIVER→ ORIGIN

When a list or order request is made without a prior alliance or required permissions.

- TYPE: 0x25
- ORIGIN: <IP:Port of the receiver>
- DESTINATION: <Name of the realm that initiated the request>
- DATA_LENGTH: Data length.
- DATA: AUTH

PING (optional for diagnostics) – ORIGIN ↔ DESTINATION

Used to check liveness if needed during Phases 2 or 3.

- TYPE: 0x26
- ORIGIN: <IP:Port of the sender>
- DESTINATION: <Nom of the destination realm>
- DATA_LENGTH: Data length.
- DATA: PING or PONG

ACK FILE - MAESTER

Maester B -> Maester A

Connection OK frame, indicating that the file transfer may begin.

- TYPE: 0x31
- ORIGIN: Empty
- DESTINATION: Empty
- DATA_LENGTH: 2
- DATA: OK

Connection KO frame, sent when the connection could not be established. The Maester will abort the file transfer.

- TYPE: 0x31
- ORIGIN: Empty
- DESTINATION: Empty
- DATA_LENGTH: 2
- DATA: KO

ACK MD5SUM - MAESTER

Maester B -> Maester A

After completing a file transfer, Maester B must verify that the MD5SUM checksum of the received file matches the original. It must respond to the sender with the following frame:

MD5SUM check successful:

- TYPE: 0x32
- ORIGIN: Empty
- DESTINATION: Empty
- DATA_LENGTH: Data length.
- DATA: *CHECK_OK*

MD5SUM check failed:

- TYPE: 0x32
- ORIGIN: Empty
- DESTINATION: Empty
- DATA_LENGTH: Data length.
- DATA: *CHECK_KO*

Erroneus frame detection procedure

If any process in the Citadel System receives a frame that does not match any of the defined formats, or has an invalid checksum, it must send back a NACK to report the error:

- TYPE: 0x69
- ORIGIN: Empty
- DESTINATION: Empty
- DATA_LENGTH: 0.
- DATA: Empty

Annex III: Guide to a good design of a project

When working on a complex system like the Citadel System, it is essential to apply good design to manage complexity efficiently and in a scalable manner. The design must have key concepts such as modularization, concurrency and robustness. As engineers, it is essential to be clear about the design before starting the implementation, since a well-structured approach will allow us to detect possible problems, optimize resources and ensure that the project is maintainable in the long term. Below are general points to consider in any C project, focusing on the important aspects of an operating system and how they can be applied to the project context.

Modularization: Design of shared modules

In complex projects such as the **"Citadel System",** properly modularizing the code is crucial to ensure clarity, reuse, and maintainability. It is not only a matter of dividing the system into independent processes such as **Maester** and **Envoys**, but also of identifying common functionalities and encapsulating them in **shared modules**. These modules, used by different processes, are essential to avoid code duplication and ensure greater efficiency.

In any project of a certain complexity, it is essential to determine which parts of the code may be common between the different processes and encapsulate them independently. If several functionalities are repeated at different points in the system, such as data manipulation or access to shared resources, it is much more efficient to organize them into reusable modules. This facilitates the maintainability of the system, as it reduces code duplication and centralizes changes: any modification to a module will automatically propagate to the processes that use it, saving time and potential errors.

Each **module** should have a **clear responsibility** and be easily reusable without modification by the processes that use it. Not only does this improve the structure of the code, but it allows **you to centralize updates** and maintain a cleaner, scalable, and more manageable system.

A good modular design in C involves separating the code into **.h files** and **.c files**. These .**h files** contain declarations of functions, constants, and data types that can be shared between modules or processes, while **.c files** implement functional logic. This separation not only makes it easier to organize the project, but also allows the compiler to better manage dependencies, which improves code integration and compilation.

When modularizing, it is important to follow these **criteria**:

1. **Reuse:** Identify the functionalities needed in different processes and encapsulate them in reusable modules. This prevents code duplication and makes the system easier to maintain.
2. **Clarification of responsibilities:** Each module must have a unique and clear responsibility. Do not mix different functionalities within the same module to avoid confusion and complicate its reuse.
3. **Independence of modules:** Modules do not have to rely heavily on each other. If there are dependencies, they must be well defined and documented to avoid future integration or maintenance problems.

Finally, it is important to avoid the use of **global variables** within shared modules. Global variables should reside only in the main processes such as **Maester**, while the modules should work only with functions that receive all the necessary arguments as parameters. This not only improves the modularity of the code, but also helps to avoid conflicts when these modules are used by multiple processes in different system contexts.

Concurrency: Getting closer to the real world
In the design of an operating system or a complex system such as the **"Citadel System",** it is essential to understand that, in the real world, many tasks must happen simultaneously to ensure the correct functioning of the system. Concurrency allows multiple users, processes, or resources to work at the same time, making the system able to respond to multiple requests efficiently.

For example, if multiple users are making requests at the same time (such as **Envoy** processes), concurrency ensures that they can be attended to simultaneously, avoiding bottlenecks and improving the overall efficiency of the system. In addition, the same process may need to receive, process, and send responses at the same time. This requires a design that allows several parallel operations to be managed without interfering with each other.

When designing a concurrent system, you can ask yourself questions to ensure that you're making well-informed decisions such as:

1. **How will you manage multiple simultaneous connections?**
   What is the best strategy for managing multiple users at once? How will you ensure that the system does not crash if the number of requests increases?
2. **How will you guarantee the robustness of the system?**
   What will happen if a thread or process fails while a request is being processed? How will you ensure that the system can continue to function without interruptions? What automatic recovery techniques can you implement to maintain service continuity?

3. **How you'll control access to shared resources**
   If multiple threads or processes need to access the same resource at the same time, how will you manage that access securely? Where do you think concurrent access conflicts might arise?
4. **How will you optimize the use of system resources?**
   Are you making optimal use of your CPU and memory? How can you make sure you're not generating more processes or threads than necessary? What impact can the mass creation of threads or processes have on the system? How to avoid resource saturation
5. **What is the best approach for each part of the system?**
   Does each part of the system need the same type of concurrent solution? Or is it possible that different parts of the system require different approaches? What would be more optimal for the different tasks that need to be performed?

The concurrent design of a system is not just about ensuring that it works properly, but that it works **well under load** and in **real-world** conditions. The key is to find solutions that balance **robustness** and **efficiency**, allowing the system to be flexible and withstand different situations. This involves continuously reflecting on the right questions during the design of each part of the system, to find the most appropriate approach based on **performance** and **stability requirements**.

As an engineer, you must be able to **critically analyze** each component of the system and adapt to the particular requirements of each process. This means looking for the best compromise between **performance**, **robustness** and **efficiency**, taking into account that what is optimal for one part of the system may not be optimal for another. Concurrency is a powerful tool, but its correct application depends on thoroughly understanding the needs of the system. The perfect solution in one context may not be applicable to other scenarios, and part of your job is to adjust and adapt these solutions to ensure that each part works optimally in its particular context.