B5 - Advanced C++

B-CPP-501

Advanced R-Type

A game engine that roars!

4.0-wip

EPITECH.

Advanced R-Type

binary name: r-type_server, r-type_client
language: C++
build tool: cmake (+ optional package manager)

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

This project of the **Advanced C++** knowledge unit will introduce you to networked video game development, while giving you opportunity to explore in-depth advanced development topics as well as to learn good software engineering practices.

Your goal: implement a multithreaded server and a graphical client for a game called **R-Type**, using an engine of your own design.

First, you will develop the core architecture of the game and deliver a working prototype, and in a second time, you will expand several aspects the prototype toward new horizons, exploring specialized areas of your choice from a list of proposed possibilities.

R-Type, the Game

For those of you who may not know this best-selling video game, which accounts for countless lost hours of our childhood, here is a little introduction.

This game is informally called a Horizontal Shmup (e.g. *Shoot'em'up*), and while R-Type is not the first one of its category, this one has been a huge success amongst gamers in the 90's, and had several ports, spin-offs, and 3D remakes on modern systems.

Other similar and well known games are the *Gradius* series and *Blazing Star* on Neo Geo.

As you now understand, you have to **make your own version of R-Type**… but with a twist not featured in the original game (nor in the remakes by the way): your version will be a **network game**, where one-to-four players will be able to fight together the evil Bydos!

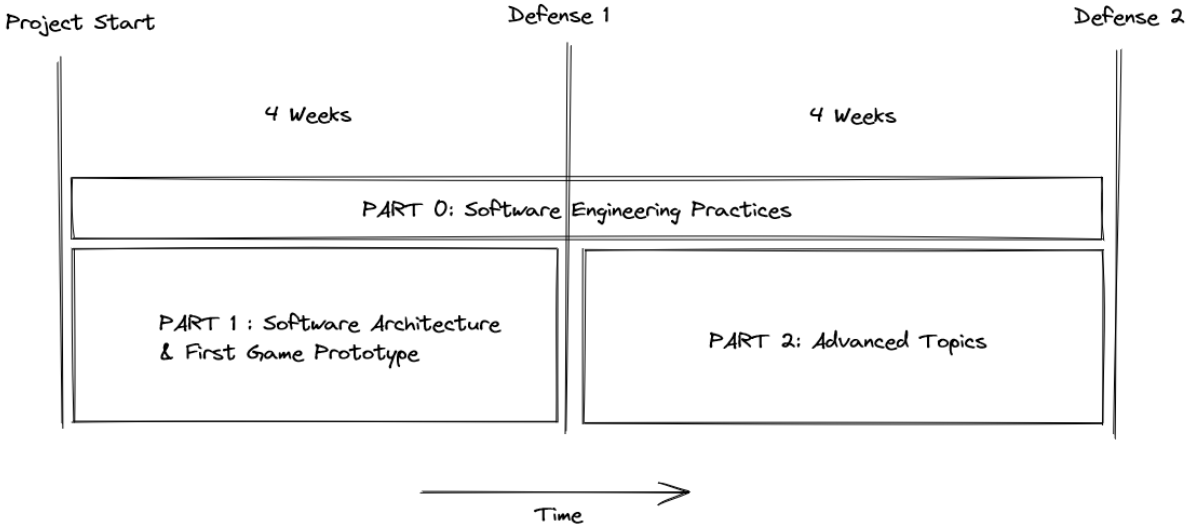
Project Organization

Please read this page thoroughly, as it is very important to understand the project schedule and organization in order to succeed.

This project is split in **two parts**, each leading to a *Delivery* and evaluated in a dedicated *Defense*.

In addition, there is also a **common part** dedicated to software engineering practices - called "Part 0" - that have to be implemented as a continuous process during development, and evaluated at both defenses.

The outline of the project is the following:



Following this organization, the outline of this document is:

- **Part 0**: Software Engineering Practices

This part details what are the expectations in terms of Software Engineering Practices your project must have. Topics such as *software documentation*, *build system tooling*, *3rd-party dependencies handling*, *cross-platform requirements*, *VCS workflow*, and *packaging* will be addressed.

These practices have to be a continuous effort, and not something done at the very end of the project. As such, each project defense will take into account the work that have been done on this topic.

- **Part 1**: Software Architecture & First Game prototype

The goal of the first part is to develop the core foundations of your networked game engine, allowing you to create and deliver a first working game prototype, or *Minimum Viable Product* (aka. *MVP*).

Obviously, this part have to be done first: you **MUST** have a working networked game to be able to continue to the second part !

The deadline for this first delivery and defense is 4 weeks after the beginning of the project.

- **Part 2**: Advanced Topics: Expand to new horizons

This second part will let you push some aspects of your MVP to a deeper level: *Advanced Software Architecture*, *Advanced Networking*, and/or *Advanced Gameplay and Game Design*.

You will have opportunity to choose which aspects you want to explore, and this will lead you to the final delivery of the project.

The deadline for this final delivery and defense is 4 weeks after the previous MVP delivery, hence 8 weeks after the beginning of the project.

General Advice

To begin with the project safely, be sure to have a good understanding of the expectations defined by the "Part 0", and focus on the **first part** first (while just taking a glance to the **second part** to let you have an idea of what may come next).

Then, once the first delivery is complete, proceed to the second part to bring your prototype to the next level !

PART 0: Software Engineering Practices

Having good Software Engineering practices is a must for any kind of serious professional project. This is how you build real products, and not just “toy projects”.

As these practices are expected to be a continuous process, items in this section are considered to be pre-requisites to the project, and will be evaluated at both defenses.

Build system, Dependencies, Cross Platform, VCS, Packaging

- The project **MUST** use **CMake** as its buildsystem. No Makefiles are allowed.
- The project **MUST** be self-contained regarding the 3rd-party libraries that may be used.

More precisely it means the project can be built and run **without** altering anything on the system: it does not rely on system-wide pre-installed libraries or development headers (for example, installed manually with *apt install* or *dnf install*), except for the standard C++ compilers & libraries and some obvious platform-specific graphical or sound system libraries (*X11*, *Wayland*, *OpenGL* implementation, *ALSA* libraries, etc.).

As a consequence, the project **MUST** use a proper method for handling 3rd-party dependencies, that could be **ONE** of the following:

- *Git submodules* ;
- *CMake Fetch_Package* or *External_Project* functionalities ;
- A dedicated package manager: *Conan*, *Vcpkg*, etc.

You choose. There is no good or bad solution, each have its advantages and drawbacks.

Copying the full dependencies source code straight into your repository is NOT considered to be a proper method of handling dependencies, even though it still allows the project to be self-contained.

In the end, the evaluator should be able to build and run your project “Out of the Box”, only using CMake and possibly some additional commands to set up the package manager.

- The project **MUST** be **Cross Platform**

It has to build and run on Windows (using Microsoft Visual C++ compiler) AND Linux (using GCC) at least.

Windows Subsystem for Linux (aka. *WSL*) is NOT a way to say a program supports Windows !

It is even better if the project can be compiled using an alternative compiler and/or platform. You'll probably discover new compiler warnings and issues you were maybe not aware of.

Examples: *Clang* on Windows or Linux, *MinGW* or *Cygwin* on Windows, *Apple CLang* on macOS

Also, you've probably already heard about *Docker*. It can be used to build and run C++ projects, and in particular, any kind of server program (including a game server). This is typically used in CI workflows.

- Adequate **VCS workflow** is expected to be used for development

You have to adopt good Git methodologies and practices: feature branches, merge requests, issues, tags for important milestones, small commits, correct commit description, etc.

Beware of excessive Liveshare usage or mob programming methodology, that could lead to poor Git usage: only one committer, ghosts contributors, no branches, etc.

It is even better if some basic form of CI/CD workflows for automation is used.

Examples: *GitHub Actions*, *CircleCI*, *Gitlab CI/CD*

Another aspect to consider is usage of specific tools such as C++ Linters, Static Analyzers, or Formatters, as they help a lot to enforce a common formatting style, or to check for additional issues not detected by compilers.

Examples: *clang-tidy*, *clang-format*

- The project can be **packaged/delivered/installed** standalone, without requiring building from sources, and with correct versioning.

This is definitively useful for end-users or enthusiasts interested in the project result, and not the source. You **MUST** provide a mean to build tarballs or installers for the game.

Examples: *CMake CPack*, or custom target in CMake using zip/tar commands

Note that final binaries shall not reference anything in the source directory, and in particular, the game assets: these have to come along with binaries.

Additionally, ready-made tarballs or installers can be made available on the project online site.

Packages build with a CI pipeline can be automatically added to GitHub release pages

DOCUMENTATION

Documentation is not your preferred task, we all know it ! However, documentation is also the first thing you'll want to look for if you needed to dive into a new project.

The idea is to provide the essential documentation elements that you'd be happy to see if you wanted to contribute to a new project.

You **MUST** write the documentation in English.

This includes:

- First and foremost, the **README** file !

This is the first thing every developer will see: this is your project public facade. So, better to have it done properly.

It has to be short, nice, practical, and useful, with typical information such as project purpose, dependencies/requirements/supported platforms, build and usage instructions, license, authors/contacts, useful links or quick-start information, and so on.

Inspire yourself from existing projects ! What do you find useful when you google for a new library or project to use ?

- The **Developer Documentation**

This is the part you don't like. But think about it: its main purpose is to help new developers to dive in the project and understand how it works in a broad way (and not in the tiny details, the code is here for this). No need to be exhaustive or verbose, it has to be practical before anything else.

Ask yourself: what you would want to know if you wanted to start development on a video game project ?

The following kind of information are typically what you'll need:

- Architectural diagrams (a typical "layer/subsystem" view common in video games)
- Main systems overviews and description (and how this materializes in the code)
- Tutorials and How-To's.

Contribution guidelines and coding conventions are very useful too. They allow new developers to know about your team conventions, processes and expectations.

Having a good developer documentation will demonstrate to the evaluator that you have a good understanding of your project, as well as the capacity to communicate well with other developers.

- The documentation have to be **available and accessible in a modern way**.

Documents such as PDF or .docx are not really how documentation is delivered nowadays. It is more practical to read documentation by navigating online through a set of properly interlinked structured pages, with a quick-access outline somewhere, a useful search bar, and content indexed by search engines.

Documentation generator tools are designed for this, allowing to generate a static website from source documentation files. Online Wikis are also an interesting alternative geared toward collaborative work.

Examples: *markdown*, *reStructuredText*, *Sphinx*, *Gitbook*, *Doxygen*, Wikis, etc.
There is many possibilities nowadays, making legacy documents definitively obsolete.

An example of well done project documentation: `https://emscripten.org`.

This have been generated from source reStructuredText files, stored in the same repository as the project's code. A CI action generates the static website from source files, then finally pushed to a server or to GitHub Pages.

- Protocol documentation

This project is a network game: as such, the communication protocol is a critical part of the system.

Documentation of the network protocol shall describe the various commands and packets sent over the network between the server and the client. Someone **SHOULD** be able to write a new client for your server, just by reading the protocol documentation.

Communication protocols are usually more formal than usual developer documentation, and classical documents are acceptable for this purpose. Writing an RFC is a good idea.

PART 1: SOFTWARE ARCHITECTURE & GAME PROTOTYPE

The first part of the project focus on building the core foundations of your game engine, and develop your first R-Type prototype (sic).

The game **MUST** be playable at the end of this part: with a nice star-field in background, players spaceships confront waves of enemy Bydos, everyone shooting missiles to try to get down the opponent.

The game **MUST** be a network game: each player use a distinct client on the network, connecting to a server having final authorithy on what is really happening in the game.

The underlying game engine **MUST** be correctly structured, with visible subsystems and/or layers for Rendering, Networking, Logic, etc.

The prerequisites for this project are explained in the previous part on Software Engineering Practices, and in particular the expectations regarding build system tooling, package manager, cross-platform requirements, packaging, and documentation.

SERVER

The server implements all the game logic. It acts as the authoritative source of game logic events in the game.

It **MUST** be multithreaded. The server **MUST NOT** block or wait for clients messages, as the game MUST run frame after frame on the server !

Your abstractions' quality will be strongly evaluated during the final defense, so pay **close** attention to them.

You **MAY** use *Asio* or *Boost.Asio* for networking, or rely on OS-specific network layer with an appropriate encapsulation (keeping in mind the need for the server to be cross-platform).

CLIENT

The client is the display terminal of the game.

It **MUST** contain anything necessary to display the game and handle player input.

It is strongly recommended that the client also run the game logic code, to prevent having too much issues due to network lag. In any case the server **MUST** have authority on what happens in the end.

You may use the **SFML** for rendering/audio/input/network, but other libraries can be used (such as **SDL** for example). However, libraries with a too broad scope, or existing game engines (*UE*, *Unity*, *Godot*, etc.) are forbidden.

Here is a description of the official **R-Type** screen:

- 1: Player
- 2: Monster
- 3: Monster (that spawns a powerup upon death)
- 4: Enemy missile
- 5: Player missile
- 6: Stage obstacles
- 7: Destroyable tile
- 8: Background (starfield)

PROTOCOL

You **MUST** design a binary protocol for client/server communications.

A binary protocol, in contrast with a text protocol, is a protocol where all data is transmitted in binary format, either raw (as-is from memory) or with some specific encoding optimized for data transmission.

Put it in other terms, prior to transmission, data is NOT converted to clear-text strings separated by end-of-line or other character (typical null-terminated _const char*_ or *std::string*). For example an integer is transmitted as-is, and not as a textual representation of the number: this is considered to be “binary” mode.

Be careful of the *Boost::serialization* library, and in particular *boost::archive::text*, as it could be used to serialize data to text data before transmission, which is not the goal you want to achieve with a binary mode protocol.

You **MUST** use UDP for communications between the server and the clients. A second connection using TCP can be tolerated, but you **MUST** provide a strong justification. In any event, ALL in-game communications **MUST** use UDP.

UDP works differently than TCP, be sure to understand well the difference between datagram-oriented (or bounded messages) communication VS stream-oriented communication.

Think about your protocol completeness, and in particular, the handling of erroneous messages. Such malformed messages or packets **MUST NOT** lead the client or server to crash.

You **MUST** document your protocol. See the previous section on documentation for more information about what is expected for the protocol documentation.

LIBRARIES

You **MAY** use the **SFML** on the client side, or other similar libraries such as **SDL** for example. However, libraries with a too broad scope, or existing game engines (*UE*, *Unity*, *Godot*, etc.) are forbidden.

The 3rd-party library scope used on the client have to be limited to: graphical, audio, input, networking and threading.

You **MAY** use *Boost* libraries on the server and/or the client. However, you have to justify each specific Boost library usage.

For networking, you **MAY** use *Asio* or *Boost.Asio* for your server and/or the client. As an alternative, on the client you may use networking functionalities provided by the 3rd-party library you use as previously described, but ONLY for the client (no SFML or similar on the server, obviously).

Asio and *Boost.Asio* are the same libraries, the former is simply a standalone version.

You may also use low-level networking functionalities provided by the Operating System.

Be aware that low-level networking APIs are generally not portable between operating systems, and as such, a strong encapsulation is needed to achieve true cross-platform, with different backends for each supported system.

As a general advice, before using a library not mentioned in this document, speak with your pedagogical team to be sure everything will be ok.

Game Engine

You've now been experimenting with C++ and Object-Oriented Design for a year. That experience means it should now be obvious to you to create **abstractions** and write **re-usable code**.

Therefore, before you begin work on your game, it is important that you start by creating a **game engine** !

The game engine is the core foundation of any video game: it determines how you represent an object in-game, how the coordinate system works, and how the various systems of your game (graphics, physics, network…) communicate.

When designing your game engine, **decoupling** is the most important thing you should focus on. The graphics system of your game only needs an entity's appearance and position to render it. It doesn't need to know about how much damage it can deal or the speed at which it can move! Similarly, a physics system doesn't need to know what an entity looks like to update its position. Think of the best ways to decouple the various systems in your engine.

To do so, we recommend taking a look at the Entity-Component-System **architectural pattern**, as well as the **Mediator** design pattern. But there are many other ways to implement a game engine ! Be creative !

General

The client **MUST** display a slow horizontal scrolling background representing space with stars, planets… This is the star-field.

The star-field scrolling must NOT be tied to the CPU speed. Instead, you **MUST** use timers.

Players **MUST** be able to move using the arrow keys.

The server **MUST** be multithreaded.

If a client crashes for any reason, the server **MUST** continue to work and **MUST** notify other clients in the same game that a client crashed.

R-Type sprites are freely available on the Internet, but a set of sprites is available with this subject.

The four players in a game **MUST** be distinctly identifiable (via color, sprite, etc.)

There **MUST** be Bydos slaves in your game.

- Monsters **MUST** be able to spawn randomly on the right of the screen.
- The server **MUST** notify each client when a monster spawns, is destroyed, fires, kills a player, and so on…

Finally, think about basic sound design in your game. This is important for a good gameplay experience.

This is the minimum, you **MUST** add anything you feel will get your game closer to the original.

PART 2: ADVANCED TOPICS: EXPAND TO NEW HORIZONS

Now that you have a working game prototype, it's time to explore new grounds and take opportunity to have a deep dive in advanced software development topics.

Three "tracks" are presented in this document, you may choose any tracks and subtopics to work on.

You are working as a Team, and a lot of the topics presented are orthogonal to each other. As such, there is definitively room for each team member to work on the second part, possibly in parallel on different topics.

So, take a deep breath, read everything in this part, discuss with your team, discuss with your pedagogical team, choose your favorite topics… and solve real-world problems !

Due to the scope of this part, bear in mind not everything have to be done to validate the project. However, it is expected some significant work to be done on one or more topics and features.

Do not start serious developments on this part until you completed the first part (first delivery and first defense) ! However, for sure you may read this part upfront to have an idea of what may come next: this can help you to set up the initial architecture of your game engine.

Advanced Architecture - Building a game engine

Nowadays, most games are built upon a "Game Engine". This track aim at extracting it from your current prototype, build upon it to improve its capabilities, then craft a whole different game using your engine.

What is a Game Engine ?

To put it simply, a Game Engine is what's left of your codebase, after you've removed the game rules, world and assets. A game engine could be specialized for a specific genre (e.g. Bethesda Creation Engine was made to build 3D RPG with branching story line), or general purpose (e.g. Unity).

All engines aim at providing a set of tools and building block to the game developers, so they can re-use common features, thus reducing development time.

Track goal

In this track, you're free to pick and choose what part of your engine you want to improve. However, we'll ask you to create another sample game to prove that your engine and the R-Type are two different projects, and we will use this new game to evaluate how re-usable your engine is. Your engine should also be separate from your R-Type and follow the same rules for self-packaging and documentation.

You don't have to implement a 2nd full-fledged game. Think of it as a demo for you engine features.

We don't necessarily want you to develop every features by yourself. You should not hesitate to ask your local unit manager to use 3rd party libraries for some advanced features (UI rendering, advanced physics, hardware support, etc.). You **MUST** abstract properly your dependencies however.

The remainder of this section will present features most engine have. Although these will be specifically looked for during the defense, additional features will also be graded.

Modularity

A good engine should not take more memory space than needed, both on a consumer disk and in memory during runtime.

A good way to achieve this is through modularization. Here are some common way to build a modular game-engine:

- **compile time**: The developer choose which module it will compile from your engine, using flags in their build system or their package manager.
- **link-time**: The engine is built as several libraries, that the developer can then choose to link with.
- **run-time plugin API**: Module are built as shared-object libraries, that are either auto-loaded from a given path or given a configuration at the start of the game, or loaded/unloaded as needed during runtime.

You **MUST** make your engine modular.

Key features

Rendering Engine
As the rendering engine is in charge of displaying information on the screen, what kind of games one can make is tightly dependent on its features (2.5D or 3D module, Particle system, Pre-made UI elements, …).

Physics engine
The physics engine primary goal is to handle collisions and gravity. More advanced engine can also allow to make entities deform, break, bounce, etc.

Audio Engine
A basic audio engine is in charge of playing background audio during gameplay. More advanced one include some SFX, from click noise in UI, to in-game noise. They can also handle positions-aware sounds in some games.

In game cinematic
An in game cinematic system takes control of the camera and in-game entities, to act through a scripted scene. Even if it could be implemented easily in the game code, it's also common to find it as an engine module.

Human-Machine interface
In most case, the engine is responsible for handling control devices, so the developer only have to specify which on they intend to use. This can go from simple feature such as setting up the keyboard and gamepad, to more advanced or integrated one, such as firing an event on a click on some UI element, supporting a touchpad, or referencing a key by its physical location instead of the letter, to better support different layout.

Message passing interface
As games are more and more advanced, the number of different systems interacting make both the synchronisation and communication more difficult to manage in a decoupled way. A message passing interface is a way to fix some of these issue, as most interaction is then handled via a system of events. Basic one allow for simple asynchronous event, while more advanced one can take priority, answer and even synchronous message when needed.

Resources & asset management
Proper resource management is a tedious task, as such it's often left to the engine, the game only referencing them by IDs or name.
Common resource management scheme include preloading (load screen), or on the fly loading, letting the engine manage a resource cache.

Scripting
In most game, entity behaviors are deferred to external scripts, as they allow for quicker test/development cycles (no need to re-compile). As such, most engine support script integration (either via custom language,

or interpreter integration).

Tools

Other than the aforementioned features, most game engine also provide some tools to the developer. These can be quality of life tools, debug or even a full-fledged IDE.

Developer console
Most computer user will know this feature in off-line game. While primarily used as a way to trigger actions, scripts or sounds during a testing phase, it's often left in the game to enable "cheating", or to help modders.

In-Game metrics
These features are most commonly used for benchmarking, or to debug specific area, and can contain a range of useful metrics such as world position, frame per seconds, lagometer (network latency, dropped frames), etc.

World/scene/assets editor
This can be standalone, or activated by a special flag at compile-time or runtime, and is used to place assets on the world, leveraging both the physics and rendering engine to be as close as possible to what would be possible during gameplay. It's also a quick-way to create new levels or to setup an in-game cinematic.

Portability

Most professional engines aim to be portables, so a game developed for one platform will work with minimal changes on another, supported by the same engine.
To achieve that, most engine will hide their implementation details behind custom abstractions the game developer will then work with.

At that point, your prototype **should** be working both on Windows and Linux. However, some further improvement could be made toward more portability.

3rd party abstractions
If you are using 3rd parties libraries, these might not be compatible with other systems (e.g. Boost might not be compatible with the latest game console, SFML is not compatible with android or iOS at the time of writing, etc.).

System features abstractions
By now you should have noticed that every system does not expose the same functions in its set of libraries (e.g. networking related functions, …).

Standard Library abstraction
Unfortunately, the c++ standard library is not always fully implemented, depending on your target system and compiler (compare the header list in freestanding and hosted libraries for example). As such, you may want to abstract some key standard library feature when it come to interaction with the target system (I/O, Networking and file system are the most common abstractions, as they integrate neatly with other quality of life features).

Advanced Server / Networking

The goal of this track is to push the server and networking subsystems of your game engine to a more mature level, matching with what is really implemented in your favorites games.

Multi-instance Server

Nowadays, most dedicated game servers are able to handle several game instances, and not only one as you have already developed up to this point.

With the possibility to have several game instances running, there is the need to bring various management functionalities (to manage game instances, users, etc…), to properly handle concurrency, as well as to provide functionalities to let users connect to the server, see the game instances, let them discuss, etc.

The idea is to bring such functionalities into your project. Here are some topics that you may investigate:

- Is the server able to run several game instances in parallel ?
- How is the concurrency handled between these instances: threads ? forks ?
- Did you have a Lobby or Rooms systems, typically used for Matchmaking ?
- How do you manage users and their identities/authentication ?
- How can they communicate between themselves and start a game ?
- Can they change some basic rules ?
- Is there some global scoreboard ?
- Do you have an administration dashboard, or a at least a text-mode console interface ? Can an admin kick/ban a specific user ?

And so on… be smart !

Advanced Networking

Your networking stack have to use UDP at this point, to be able to investigate this topic.

Due to inevitable network issues, such as low bandwidth, latency/lag, or unreliability (packet losses, out of order packets, duplication), the game might face some serious problems leading to suboptimal gameplay experience, or worst, full desynchronizations or crashes.

The general question to answer on this topic is what mechanisms are you providing to help mitigate on those network issues, without sacrificing performance.

Here is some useful tools to evaluate the behavior of the program under degraded network conditions (simulate lag, packet losses, duplication, reordering, low bandwidth, throttling, etc.):

- on Windows: clumsy
- on Linux: netem

Be sure to test your game under degraded network conditions. Using one of these tools, what is the behavior of your game with:

- A 2% packet drop, 2% packet duplication and 5% packet reordering ?
- A latency > 150ms ?
- A very low bandwidth ?

You'll be probably surprised by the results.

Note that these tools might be used during the Defense, to verify your claims about successfully handling network issues.

Here are some technical aspects that you may investigate:

- **Data packing**

If plain in-memory data is sent as-is over the network this can be really inefficient. Usual layout of data in memory is not necessarily optimized for such transmission, because of primitive data types that may be too large, internal padding of structure fields optimized by the compiler for CPU alignment, etc.

Hints: Data types sizes, bit-level data packing, structs alignment and padding optimization, data quantization, etc.

- **General-purpose data compression**

In data transmission there is generally a lot of redundancy. Using general purpose encodings and compression algorithms is a must to reduce bandwidth usage.

Hints: RLE (Run Length Encoding), Huffman encoding, zlib-like algorithms and libraries

Side question:
- Do you know what is the average bandwidth used by your game ?

- **Network errors mitigation techniques (packets drops, reordering, duplication)**

UDP protocol is unreliable, and in case of heavy network congestion packets may be lost, reordered, or even duplicated. The game might suffer from these issues, giving weird results, or worst, complete desynchronizations.

You should provide means to prevent any kind of issues caused by UDP unreliability: packet drops, reordering, and duplication.

Hints: datagram sequence number, fault-tolerance

Side questions:
- Do you know what is your average UDP datagram size ?
- Have you already heard about MTU and packet fragmentation ?

- **Message reliability**

Following the previous point, even though some UDP datagrams might be lost, various messages MUST be sent/received reliably (example: connection to a game instance, a player is dead, etc.).

Do you provide a technical solution for this need ?

Hints: dedicated TCP channel for reliable message delivery, “ACK” patterns for UDP, message duplication

Lag Compensation

The most dreaded issue in game network programming is **latency**, or more familiarly, *lag*.

Latency is measured by the **ping**, or round-trip time for a message to be sent and replied between a client and a server. On local networks, with typical low latency (< 25ms), this is generally not really an issue. Over the internet, latency can become very high (>100ms), and worse, it can be extremely variable.

The consequences of latency are what you regularly observe in your favorite network game sessions: players and entities seems to “jump” positions erratically, strange input delay lag, or being killed while hidden behind a wall, etc.

The general problem is that due to network latency, not everyone is aware of what’s happening at the same time. As such, things might get desynchronized at some point (client A, server and client B each have different information / game state), or some long delays might be observed between your actions and their immediate results.

However, the game developers have to give the illusion everything is **smooth** and **consistent**. Tricky topic !

Example techniques: *client-side prediction*, *server reconciliation*, *input delay*, *rollback / replay*, *entity state interpolation*, *remote entities extrapolation*, *deterministic simulation*, etc.

There is many possibilities to address this issue, not necessarily mentioned in this document. As a convenience, an annex document is provided along with this document, giving some additional materials, pointers and external links to dive into this topic.

Advanced Gameplay / Advanced Game

The goal of this track is to enhance the Gameplay and Game Design aspects of your game.

Up to this point, your R-Type should be a working prototype with a limited set of functionalities and content gameplay wise. Let's change this, and make your game more enjoyable for final players AND for game designers !

Players: elements of Gameplay

While R-Type is an old game, it is nevertheless feature-complete game: with many monsters, levels, weapons, and other gameplay elements. Can you move your prototype to the next level ? It should be **fun** to play.

Elements you might consider:

- Monsters, with varying movement patterns and attacks. Example: snake-style monsters as in level 2 of R-Type, or ground turrets.
- Levels, with different themes, and interesting gameplay twists. Look at the level 3 of original R-Type: you are fighting with a huge vessel during the whole level, or level 4 with monsters leaving solid trails behind them.
- Bosses. They are legendary in the R-Type lore, and played a significant role in the game success. In particular the idiomatic first boss, frequently pictured on the game boxes: the *Dobkeratops*.
- Weapons. For example, the **Force** plays an integral part in the original game: it can be attached to front or back of the player (and when attached back, it shoots backwards), can be "detached" and acts independently, can be re-called back, it protects the player from enemy missiles, allow to shoot super-charged missiles, etc.
- Gameplay rules: is it possible to change or tweak the game rules ? Think about things such as friendly-fire, bonuses availability, difficulty, game modes ("coop", "versus", "pvp", etc.)
- Sound design: it plays an integral role in any gameplay experience.

Inspire yourself from the existing game:
- Gameplay overview
- Individual Stage breakdown

The grading will not evaluate only quantitatively. E.g. having N or M kind of element is not the only things that matters. Having reusable subsystems to add content easily is equally important. See the next point.

Game Designers: content creation subsystems

Having a lot of content is great, but having great subsystems allowing to create easily new content is even better.

Do you provide such systems to allow Game Designers - **which are not necessarily seasoned game developers** - to easily add new content into the game ? Do they have to know C++ and modify many files of your project (possibly having to recompile everything when they add a new level or boss for example) ?

Your game engine **SHOULD** provide well-defined APIs allowing for runtime extensibility (e.g. plugin system, Dlls) to add new content, standard formats, or even a dedicated scripting language to program entities behaviors (read: simpler than C++!).

Example: Lua programming language is frequently use in video games.

The general question to answer is: how easy it is to add new content and behaviors in the game ? Of course, knowing C++ can be a requirement, but does your system simple enough to use on a daily basis by people with limited development skills (but potentially with great Game Design skills) ?

Even better, do you have any content-editor tools for various game elements (level-editor, monster editor, etc.) ? Those could be separate programs, or embedded directly in the main game.

Documentation is a critical part of any content creation tool, API, or system. Tutorials and How-To's in particular are definitively what content creators are looking for.

This part is equally important than the previous one: there is no point having 5 levels if each level require heavy modification of internals of the base game. Better to have only 2 levels, and demonstrate to the evaluator that your subsystem can be used to add any level you want easily.

Single Player Game

This project focused on multiplayer game. But you may also want to play a solo version of the game.

Does your architecture allow this ? Maybe you would want to spawn automatically a local server to do this, or re-introduce game logic in the client.

Playing the solo version does not necessarily mean you are alone: having Bots for other players could be a very good idea. Doing an AI for a R-Type game can also be something very interesting to do !