B5 - Advanced C++

B-CPP-500

R-Type

A game engine that roars!

4.1

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, and will give you the opportunity to explore advanced development techniques as well as to learn good software engineering practices.

The goal is to implement a multithreaded server and a graphical client for a well-known legacy video game called **R-Type**, using a game 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 to the next level, exploring specialized areas of your choice from a list of proposed options.

R-Type, the Game

For those of you who may not know this best-selling video game, here is a little introduction.

This game is informally called a Horizontal Shmup (or simply, a *Shoot'em'up*), and while R-Type is not the first one of its category, it 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*.

In this project, you have to **make your own version of R-Type**, with additional requirements not featured in the original game:

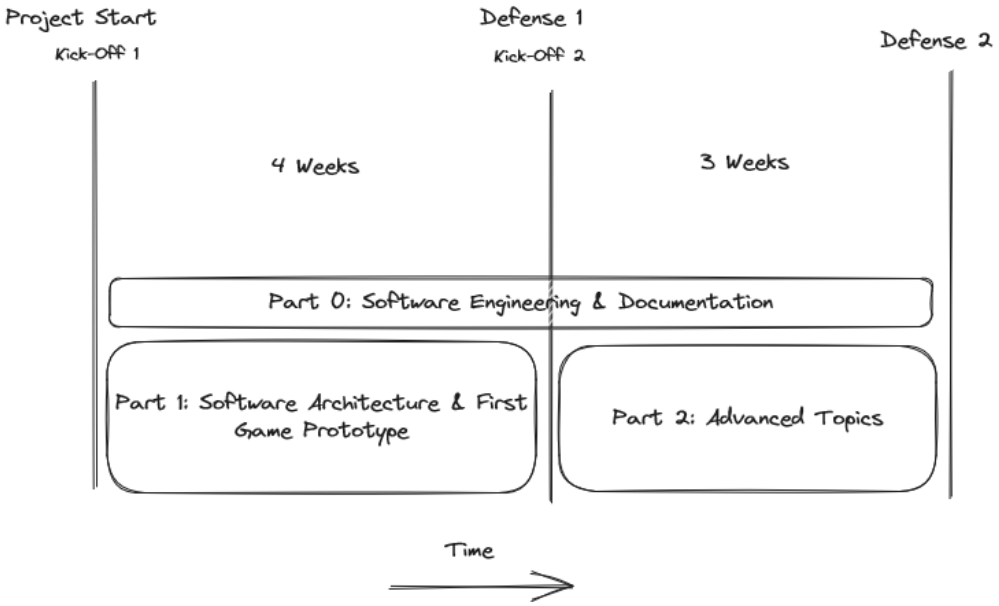
- it **MUST** be a **networked game**, where one-to-four players will be able to fight together the evil Bydos;
- its internal design **MUST** demonstrate **architectural patterns of a real game engine**.

Project Organization

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

Additionally, there is also a *common part* - called "Part 0" - dedicated to **Software Engineering** and **Documentation**. This has to be implemented as a continuous process during development, and will be evaluated at both defenses.

The outline of the project can be summarized with the following diagram :



This document is structured following this diagram decomposition :

- **Part 0**: Software Engineering and Documentation Requirements

This part defines the expectations in terms of Software Engineering and Documentation practices your project must have. Topics such as *technical documentation*, *build system tooling*, *3rd-party dependencies handling*, *cross-platform support*, *version control 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 and software architecture of your networked game engine, allowing you to create and deliver a first **working game prototype**.

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

- **Part 2**: Advanced Topics : from game prototype to infinity and beyond!

The goal of this second part is to enhance different aspects of your prototype, bringing your final delivery to a more mature level. There is 3 technical tracks you may want to work on: *Advanced Software Architecture*, *Advanced Networking*, and/or *Advanced Gameplay and Game Design*.

You will have opportunity choose which topics you want to work on, finally leading you to the final delivery of the project. The deadline for this final delivery and defense is 3 weeks after the first delivery, for a total of 7 full weeks for the whole project.

General Advice

This is probably the first project you have with a subject of a substantial size. Things start to get serious now !

So, to begin with the project safely, be sure to have a good understanding of the expectations defined by the “Part 0”, and start working on the **first part first**, while just taking a quick 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 and Documentation Requirements

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.

Software Engineering Requirements

- The project **MUST** use **CMake** as its build system. No Makefiles are allowed !
- The project **MUST** be **fully self-contained** regarding its dependencies and 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 low-level and platform-specific graphical or sound system libraries such as OpenGL/X11/etc.

In particular, usage of SFML, SDL or any other kind of high-level 3rd-party library **MUST** be properly packaged along with the project, both at compilation time and runtime.

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 or Vcpkg.

Copying the full dependencies source code straight into your repository is NOT considered to be a proper method of handling dependencies !

The evaluator should be able to build and run your project "Out of the Box", only using its locally installed C++ compiler and environment, the CMake tool, and a shell.

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

It has to build and run on Windows using Microsoft Visual C++ compiler **AND** Linux using GCC.

Windows Subsystem for Linux (aka. *WSL*) is **NOT** considered to be a way to achieve true cross-platform.

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: *LLVM Clang* on Windows or Linux, *MinGW* toolchain, *Apple CLang* on macOS

Additionally, you may also enable compilation and execution in Docker. While mostly used in Web projects, Docker can also be used to build and run C++ projects, and in particular, any kind of server program such as a video game server.

- A well-defined **Version Control System workflow MUST** be used for development

You have to adopt good Git methodologies and practices. You must demonstrate good usage of feature branches, merge requests, issues, tags for important milestones, commits content and description, etc.

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

It is even better if a CI/CD workflow is used, to build, test, and even deploy the server.

Examples: *GitHub Actions*

Another aspect to consider is the usage of specific tools such as C++ *linters* or *formatters*, as they help a lot to spot common programming errors, and enforce a common programming style in a team.

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

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

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.

Documentation Requirements

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 yourself to a project for a new team.

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.
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.

Note that generating documentation from source code comments like with the *Doxygen* tool, while it is a good practice, can't be considered **alone** to be a real project documentation. You **MUST** produce documentation at a **higher level** than only just classes/functions descriptions !

- 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.

No platform specific tool shall be needed to read the documentation. Any user should be able to read the documentation regardless of the system used. As so, a Web delivery is the most appropriate way to publish documentation.

- 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 general goals are the following:

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

- The game **MUST** be a networked game: each player use a distinct Client program on the network, connecting to a central Server having final authority on what is happening in the game.

- The game **MUST** demonstrate the foundations of a game engine, with at least visible and decoupled subsystems/layers for Rendering, Networking, and Game Logic.

Server

The server implements all the game logic. It acts as the **authoritative** source of game logic events in the game: whatever the server decides, the clients have to comply with it.

On a typical and simplified client/server video game architecture, the Clients sends local user inputs to the Server, which processes them and update the game world, and send back regular game updates to all Clients. In turn the Clients renders the updated game world on the screen.

There is however many variations around this basic principle, and you have to design your solution.

- The server **MUST** notify each client when a monster spawns, moves, is destroyed, fires, kills a player, and so on…, as well as notifies others clients' actions (a player moves, shoots, etc.).
- The server **MUST** be multithreaded, or at least, **MUST NOT** block or wait for clients messages, as the game must run frame after frame on the server regardless of clients' actions!
- 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. More generally, the server must be robust and be able to run regardless of what's wrong might happen.
- You **MAY** use the *Asio* library for networking, or rely on OS-specific network layer with an appropriate encapsulation.

Asio and *Boost.Asio* are the same libraries, the former is a standalone version and the latter come as a part of the Boost libraries.

If you decide to use OS-specific networking API, keep in mind the need for the program to be cross-platform. You'll need to create network encapsulations to allow for real cross-platform code.

Note that your abstractions' quality will be evaluated during the final defense, so pay close attention to them.

Client

The client is the *graphical display* of the game.

It **MUST** contain anything necessary to display the game and handle player input, while everything related to gameplay logic shall occur on the server.

The client **MAY** nevertheless run parts of the game logic code, such as local player movement or missile movements, but in any case the server **MUST** have authority on what happens in the end. This is particularly true for any kind of effect that have a great impact on gameplay (death of an enemy/player, pickup of an item, etc.): all players have to play the same game, whose rules are driven by the server.

You may use the **SFML** for rendering/audio/input/network, but other libraries can be used (such as **SDL** or **Raylib** for example). However, libraries with a too broad scope, or existing game engines (*UE*, *Unity*, *Godot*, etc.) are strictly 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 as-is from memory (raw data) or with some specific encoding optimized for data transmission.

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 communication VS stream-oriented communication.

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

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

Game Engine

You've now been experimenting with C++ and Object-Oriented Design for a year. That experience means you should be able 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 prototype **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 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 ! Think of the best ways to decouple the various systems in your engine.

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, from common Object-Oriented paradigms to full Data Driven ones. Be sure to look for articles on the Internet.

Gameplay

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

The star-field scrolling, entities behavior, and overall time flow must NOT be tied to the CPU speed. Instead, you **MUST** use timers.

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

There **MUST** be Bydos slaves in your game, as well as missiles.

Monsters **MUST** be able to spawn randomly on the right of the screen.

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

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

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

This is the minimum, you can 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, each divided in different topics. 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 completely unrelated features.

So, take a deep breath, read everything in this second 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.

Track #1: Advanced Architecture - Building a real game engine

Nowadays, most games are built upon 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 Goals

In this track, the goal is to continue the design of your game engine, in order to deliver a well-defined engine, with proper abstractions, decoupling, features and tools.

The next document section presents features most engines have.

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.)

In a second step, you'll want to make the game engine a generic and reusable building block, **delivered as a separate and standalone project**, completely independent of your original R-Type game.

That is, your R-Type game will be using the engine as a library dependency, while the engine itself knows nothing about anything related to the game logic/assets/functionalities of R-Type.

Note that the standalone engine should follow the same rules for packaging and documentation as the rest of the project.

To demonstrate that your game engine is fully generic and standalone, the ultimate goal is to **create a 2nd sample game** (different from R-Type !), using your standalone game engine. With this, you can prove it is really a **reusable and generic system**, on which you can build various games on top of it.

You don't have to implement a 2nd full-fledged game to validate this goal. Think of it as a demo for you engine features. However, the quality of the 2nd game will be evaluated: from a basic pong game, to a full-fledged completely different game like a Mario clone, and all the variations in-between.

Game Engine Features

Here is a list of the most essential features a solid game engine might have. The quantity and quality of subsystems and features in your engine will be evaluated, as well as anything additional and relevant not mentioned in the following list.

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 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.

Engine Subsystems

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).

LUA or *Python* are typically used as scripting systems

Engine Tooling

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 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. It also introduces the concept of customizable Console Variables (aka. "CVars").

In-game metrics and profiling
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, resource usage (CPU, Memory), Frame per Seconds (FPS), or 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 set up an in-game cinematic.

The quality of the editor(s) will be evaluated: from a very simplistic solution using only configuration files, to a fully fledged interactive drag-and-drop graphical application.

Track #2: Advanced Networking - Building a reliable network game

The goal of this track is to enhance the server and networking subsystems of your game engine, matching with functionalities actually implemented in your favorites games.

This track offers 3 topics you might want to work on: *Multi-instance Server*, *Data Transmission Efficiency and Reliability*, and *High-level Networking Engine Architecture*.

Multi-instance Server

Most dedicated game servers are able to handle several game instances in parallel, 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 functionalities to manage them, handle the concurrency, provide methods to let users connect to the server, see the instances, join the games, discuss, etc.

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

- The ability for the server to run several different game instances in parallel;
- A Lobby/Room system, typically used for matchmaking, or simply instances discoverability;
- Users and identities management: storage, sessions, authentication, etc.;
- Communication between users: text chat, or even voice;
- Game rules management: change some basic game rules, per game instance;
- Global scoreboard and/or ranking system;
- Administration dashboard: text-mode console interface (example: an admin can kick/ban a specific user), or even web-based interface.

Data Transmission Efficiency and Reliability

In your current prototype the Server is probably sending updates to each client 60x/seconds for all entities in the world, without deeper technical consideration. While on a local network it should offer a reasonable experience, over the Internet this might be a quite different story.

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

So, the general question to answer on this topic is: what mechanisms can you provide to help mitigate on those network issues ? Here are some of the common topics that you may want to investigate:

- **Data packing**

Plain in-memory data sent “as-is” over the network can be really inefficient, even if using a binary protocol. The usual layout of data in memory is not necessarily optimized for data transmission, because of primitive data types that may be too large, internal padding of structure fields optimized by the compiler for CPU alignment and not for data transmission, etc.

Hints: adequate data types sizes, space-efficient serialization, bit-level 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 an interesting way to reduce bandwidth usage. Additionally, a few compression techniques are tailored specifically for games.

Hints: for general purpose algorithms, *RLE* (Run Length Encoding), *Huffman encoding*, *LZ4*, *zlib*. For games-specific techniques: *delta snapshot compression*.

Some compression algorithm are more CPU intensive than others, hence less suited for real-time network gaming

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

The UDP protocol is unreliable, and in case of heavy network congestion, packets may be lost, throttled, reordered, or even duplicated. The game might suffer a lot from these issues. You should provide means to prevent any kind of issues caused by UDP unreliability.

Hints: UDP datagram size vs. MTU, datagram buffers and sequences numbers, fault-tolerance functionalities.

- **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.).

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

High-level Networking Engine Architecture

Fast-paced networked video games such as R-Type or any FPS are fundamentally **real-time distributed simulations** over a network.

The central design issue of such programs is that their state (i.e. the "game world" in our case) needs to be consistent across each individual participant of the simulation (i.e. Clients and the Server), in real-time, **BUT** there is an incompressible communication **latency** between them, familiarly called "lag".

As such, any information sent, such as a player move or a game update, will be received by others slightly **in the future**, and any information received from others will come slightly **from the past**. Basically, it means that at any given point in time, each participant have a slightly different version of the world than the others.

Yet, the mission of the game developers is to give the illusion everything is happening in a **consistent** and **efficient** way… Tricky topic!

This paradox explain what you regularly observe in your favorite network game sessions: players and entities positions are sometimes unstable or "teleports" inconsistently, there can be some annoying input lag or lack of responsiveness, or more frustrating, you are sometime killed while hidden behind a wall.

- **Possible solutions**

As our architecture is based around a central Authoritative source of truth, the Server, there is several possible solutions to address the problem.

- At one side, you can decide for Clients to stop their own time and always wait for last updates of the world from the Server. The Server is then the central orchestrator for all Clients.

This allows for **full consistency**, and is probably what you have done up to know. However, this can lead to **serious performance problems** in case of degraded network conditions. Also, it will come at the cost of high CPU and bandwidth usage on the Server: in order for your game to be responsive, the Server needs to send updates at a very high frequency (60x/seconds at least) for each connected client.

- At the opposite side, you can decide for Clients to not wait at all for the Server and run their own game simulation regardless what is happening elsewhere, letting them figuring out how to integrate the game updates they receive "from the past" in their own timeline.

This leads to **high performance**, but this can greatly **degrade consistency** as each participant will have slightly different view of the world. Accumulation of such divergent state over time, if not corrected properly, would lead to inconsistent gameplay experience, or worst, complete desynchronizations.

As usual, a pragmatic solution lies in-between these two opposite approaches. The goal here is to find the best balance between **game state consistency** and **networking performance**. The following hint box gives some keyword to look for, about various techniques usually implemented in networked video games.

Hints: **client-side prediction** with **server reconciliation**, low frequency server updates with **entity state interpolation**, **server-side lag compensation**, *input delaying*, *rollback netcode*.

Advanced Gameplay - Building a Fun and complete video 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. You should move your prototype to the next level gameplay-wise. It should be **fun** to play!

Elements you might consider:

- Monsters, with varying movement patterns and attacks. Example: the **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 interesting to let users change or tweak the game rules. Think about things such as friendly-fire, bonuses availability, difficulty, game modes ("coop", "versus", "pvp", etc.)
- Sound design. This plays an integral role in any gameplay experience: music (or even better, procedural music), environmental effects, sound effects, etc.

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

The grading will not evaluate only quantitatively, but also qualitatively: 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 section.

Game Designers: content creation tools

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

Game Designers are not necessarily seasoned developers, and it is very useful for them to be able to easily add new content into the game. The need 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) is generally a showstopper.

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.

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 the system should be simple enough to be used on a daily basis by people with limited development skills.

It is even better if you have **content editor tools** for various game elements (level-editor, monster editor, etc.). Those could be separate programs, or embedded directly in the main game. See the related section in the "Advanced Architecture" track for more information on this topic.

In all cases, having solid and useful tools or content creation subsystems is equally important to having a lot of content: there is no point to have 5 levels if each level require heavy modification of internals of the base game. It is better to have only 2 levels, and demonstrate to the evaluator that your system can be used to add any level you want easily.

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

Single Player Game

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

Your current architecture might need to be adapted. Maybe you would want to spawn automatically a local server to do this, or re-introduce game logic in the client, etc.

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