To appear in proceedings of EuroPlop 2025

Netcode Patterns for Real-Time Online Multiplayer Games in Unity

Simon Furrer[1][0009−0004−0806−4508], Alexandre Bergel[2][0000−0001−8087−1903], and Timo Kehrer[1][0000−0002−2582−5557]

[1] University of Bern, Bern, Switzerland
[2] RelationalAI, Switzerland/USA

Abstract. With the rapid growth of the gaming industry, real-time online multiplayer games have surged in popularity, driving an increasing demand for the development of complex software. In particular, game developers often face significant challenges when working with networked code (netcode). The design space of available netcode libraries is large, yet there is limited guidance when implementing an online multiplayer game. To compensate, we present a catalog of netcode patterns providing guidance when implementing real-time strategy and role-playing games. We use two concrete games serving as case studies for pattern identification and validation.

Keywords: Games · Online · Multiplayer · Netcode · Patterns · Unity · Netcode for GameObjects · RTS · RPG.

1 Introduction

In 2024, the gaming industry generated more than $180 billion in revenues with an audience larger than 3 billion people, and the global video game market is expected to reach more than $213 billion in 2027 [24]. This trend has intensified the demand for dedicated software engineering principles to support increasing complexity. On the one hand, research on software engineering for games has tackled mostly analytical aspects such as automated testing and debugging [6,9,28,30]. On the other hand, practitioners seek for guidance in game programming and design, as addressed, e.g., in Robert Nystrom's widely recognized textbook on *Game Programming Patterns* [27]. However, while the book covers a mix of classical design patterns and game-specific ones that became standard in today's game engines, these patterns do not address the challenges one faces when developing the *networked* aspects of *online multiplayer games*.

In this paper, we focus on *game engine APIs dealing with networked code (netcode)*, which emerged to be a standard technology stack in *real-time online multiplayer game development*. While netcode libraries provide a general basis to implement networked game logic, they keep a lot of design freedom, and their targeted design space is large. Navigating the solution space and balancing design flexibility with ease of use is challenging.

Inspired by Nystrom's book [27], but with a much more specialized focus, we argue that *netcode patterns* can serve as a valuable solution that tackles this lack of guidance. We focus on real-time online multiplayer games implemented on top of Unity's *Netcode for GameObjects (NGO)* [15], our primary target audience is game developers who want to create their first online multiplayer game in Unity.

Throughout the paper, we use two concrete games serving as case studies for pattern identification and validation: Unity's online multiplayer Role-Play Game (RPG) known as Boss Room [35] as well as a Real-Time Strategy (RTS) game called The Shroomlings specifically developed for this paper. We present an extensible catalog of nine netcode patterns for multiplayer RTS and RPG games, classified into four groups: *Organizational patterns* primarily address the question of whether the server or the client holds authoritative control, establishing the source of truth for game state and logic; *consistency patterns* focus on maintaining a consistent game state across all clients. *Security patterns* aim to prevent players from cheating. *Serialization patterns* deal with reducing bandwidth usage and minimizing the effort required to serialize and transmit objects over the network. Orthogonal to our primary categorization, the patterns range from high-level architectural patterns to concrete design and implementation patterns. The patterns are documented in EuroPLoP's well-established format, providing a name along with five sections: *Context*, *Problem*, *Forces*, *Solution*, and *Consequences* (positive and negative) [8,13,33].

Section 2, intended primarily for newcomers and optional for experts, outlines key concepts essential for understanding online multiplayer games and their associated design challenges. Section 3 provides an overview of our overall methodology and presents the two investigated games. Section 4 documents our pattern catalog, while Section 5 discusses potential limitations. In Section 6, we review related work in a broader context, and Section 7 concludes the paper along with an outlook on future work.

2 Background

In this section, we introduce the fundamentals of what defines a game and its implementation on top of Unity and NGO, highlighting major challenges when developing real-time online multiplayer games.

2.1 The Game Loop

A game can be seen as an interactive video, which immerses the player in a virtual world. The player can provide inputs such as keyboard and mouse events affecting the video. A video is a sequence of images called frames, which must be computed. The *game loop* is a continuously repeating process that drives frame updates during gameplay. It consists of several stages that process game events, update game objects - the fundamental object type in Unity used to represent everything in your project - and render the game graphics. The game loop is responsible for keeping the game running at a consistent frame rate and ensuring

that the game logic and visuals are synchronized. Typical stages of a game loop are "Initialization," "Physics," "Player input," "Game logic," "Rendering," and "Decommissioning". To add custom code specific to the game, developers must implement hooks that get called according to the game loop. Unity provides a detailed image showing the script lifecycle flowchart with all hooks and the order in which they get called by the game loop [32].

2.2 Unity's Netcode for GameObjects

Netcode refers to an abstraction layer that simplifies everyday networking gameplay needs and integrates with gaming services such as game server hosting and monetization. One of the key companies in the global game engines market is Unity Software Inc., with their widespread game engine Unity [12] which is compatible with several netcode solutions, providing a complete service ecosystem. On a conceptual level, a flowchart provided by Unity illustrates all hooks and the sequence in which they are invoked by the game loop [32]. On an implementation level, Unity has two different first-party netcode solutions being actively maintained: Netcode for GameObjects (NGO) and Netcode for Entities (NE) [15]. For the work of this paper, NGO has been chosen as it has less overhead than NE. The following concepts [31] of the NGO API are of particular relevance for this paper.

`NetworkObject` is a fundamental component that enables game objects to be synchronized across multiple clients in a multiplayer game. `NetworkBehaviour` is primarily used to create unique game logic. It can also be used to synchronize state and send messages over the network. User scripts extend `NetworkBehaviour` to implement networked behavior. `OnNetworkSpawn` and `OnNetworkDespawn` are user callbacks being invoked on each NetworkBehaviour. The former belongs to the "Initialization" stage of the game loop, while the latter belongs to the "Decommissioning" stage. The patterns documented in this paper also make use of the hooks `Awake` (an initialization hook) and `Update` (executed every frame of the game loop). `NetworkVariable` provides a mechanism for persistently synchronizing properties between servers and clients, meaning also late-joining clients receive the latest state. `NetworkVariable` ensures continuous data synchronization. Remote procedure calls (RPCs) allow calling methods on objects that are not in the same memory space. In this paper, we use `ServerRpc` for a client to invoke a procedure on a `NetworkObject`, which is then executed on the server version of the same `NetworkObject`. We use `ClientRpc` for a server to invoke a procedure on a `NetworkObject`, that is then executed on all clients' version of the same `NetworkObject`. The `NetworkManager` component comprises all of the project's netcode-related settings. It must be a singleton and can be seen as the central netcode hub for a netcode-enabled project. `NetworkTransform` manages the complexities of synchronizing a game object's transform (3D position, rotation, and scale) across the network, including optimizing bandwidth usage for transform updates. This component also ensures that late-joining clients receive the latest transform state. `NetworkAnimator` provides a fundamental example of

how to synchronize animations during a network session. This component also accounts for late-joining clients.

2.3 Challenges in Online Multiplayer Game Development

Game development must address the dual challenges of ensuring Quality of Service (QoS) and Quality of Experience (QoE) [21]. These challenges are interrelated and require distinct considerations to balance technical performance with player satisfaction. Depending on the game genre, there might be different QoS and QoE factors of importance [21]. For real-time online multiplayer games, the following factors crucially contribute to the overall QoS and QoE:

1. *Consistency* ensures that simulations on clients align with what should occur in subsequent frames. The goal is to avoid desynchronizations, e.g., when clients disagree on the state of the game, leading to discrepancies such as an object's position differing across clients.
2. *Responsiveness* refers to the time it takes for the game to provide feedback after a user action. High responsiveness is crucial for maintaining an immersive and engaging player experience.
3. *Security* prevents cheating to sustain player trust and enjoyment. Online multiplayer games must be protected against manipulation of client-side code, such as forged network messages sent to the server, through which players can exploit vulnerabilities to gain unfair advantages.

Achieving optimization across all three properties is particularly difficult due to the distributed nature of online multiplayer games. Inherent latency and limited bandwidth of network communication must be carefully managed to provide a seamless and secure gaming experience, often leading to tradeoffs between consistency, responsiveness and security [20].

3 Methodology

The patterns proposed in this paper are based on the design of two concrete games serving as case studies. The first one involves extracting pattern candidates by analyzing an existing real-world game known as Boss Room [35], providing insight into practical, field-tested solutions. The second one is a Real-Time Strategy game called The Shroomlings which has been developed specifically for this paper, allowing for controlled exploration and validation of pattern candidates derived from Boss Room.

We followed a qualitative approach as outlined in EuroPLoP's pattern introduction guide [8], employing an iterative process supported by a review and feedback cycle [33]. The initial identification and description of patterns were provided through code analysis by the first author of this paper, who has over 10 years of experience developing games with Unity. The patterns were subsequently validated and refined based on feedback from the second and third author.

Fig. 1. A "Boss Room" session [34].

Fig. 2. A "Shroomlings" session.

To identify the "Known Uses" for each pattern within the case study games, we followed a quantitative approach counting pattern instances to get a better impression of how frequently a pattern is actually used. While counting pattern instances was straightforward for The Shroomlings, which has been fully implemented by the first author, this was more challenging for the substantially larger Boss Room. Given that detecting pattern instances is hardly a straightforward mechanical task [23], we followed a lightweight yet semi-automated process. In a first step, we used full-text search to locate occurrences of NGO API concepts present in the code snippets documented within the pattern. Second, we validate each occurrence by comparing its context in the project code to the documented pattern, yielding a verdict of whether the found occurrence represents an actual pattern instance or not.

3.1 Analysis of Existing Game Code

The first source of information about how online multiplayer games can be successfully developed was existing game code. Unity provides the full game code of "Boss Room", a cooperative multiplayer Real-Time Strategy developed using Unity Netcode. It is an educational sample designed to showcase typical netcode solutions often featured in similar multiplayer games [34,35]. In "Boss Room," players control a character from a third-person camera perspective (Fig. 1). The objective is to collaborate with teammates to defeat goblin enemies and ultimately overcome a final boss. This paper considers version 2.2.0 of the game.

3.2 Developing a New Game From Scratch

The second case study has been developed specifically for this paper, allowing for controlled exploration and validation of pattern candidates. Given that the primary author has over 10 years of experience in game development with Unity, a new version of the single-player game design for "The Shroomlings" was built from scratch, incorporating multiplayer features. "The Shroomlings" is a competitive Real-Time Strategy game in which players control a shroomling village competing against another village (Fig. 2).

Table 1. Overview of the netcode patterns.

	Organizational	*Consistency*	*Security*	*Serialization*	*Architectural*	*Design/Impl.*
Server First	*		*		*	
High-Level Branching	*					*
Low-Level Branching	*					*
Authoritative Branching	*					*
Continuous Synchronization		*				*
One-Shot Synchronization		*				*
Sanity Check			*			*
Marshaling by Reference				*		*
Marshaling by Value				*		*

4 Netcode Patterns

In this section, we document the identified netcode patterns in the form of EuroPLop's pattern introduction guide [8,13,33]. Each pattern is illustrated by an example taken from the The Shroomlings. Table 1 summarizes the documented patterns. We primarily organize the patterns based on their purpose, categorizing them as organizational, consistency, security, and serialization patterns. Orthogonal to our primary categorization, the patterns range from high-level architectural patterns to concrete design and implementation patterns. Note that Server First can be categorized as both organizational and security pattern.

4.1 Server First

Context. When implementing networked behaviors in online multiplayer games, which require synchronization, you must decide on authoritative control, establishing the source of truth for game state and logic.

Problem and Forces. How can you decide whether to implement synchronization server- or client-authoritative while optimizing for consistency, responsiveness, and security? You would like to achieve security and consistency first and responsiveness second. Furthermore, adhering closely to NGO's API principles is essential for maintaining compatibility.

Solution. Default to server-authoritative.

Example. Consider synchronizing unit spawns. When the player clicks to spawn a unit, the action shall be validated, and units shall be spawned on each client in a reasonable amount of time. You should send the player's inputs to the

server, validate the action, and, if valid, spawn the units on each client. With this, you achieve security, consistency, and a reasonable response time through a server-authoritative approach.

Consequences. This pattern has the following benefits: (B1) Security and consistency by default. These two aspects are usually more critical than responsiveness for Real-Time Strategy games. (B2) Easier usage of NGO's API since it is server- authoritative by default. The pattern's liabilities are as follows: (L1) Authority alone always means a tradeoff between consistency and security versus responsiveness; therefore, tradeoff is needed to solve a complex situation.

Known Uses. This pattern can be found in two games, Boss Room and The Shroomlings. The documentation of the former states that "the nature of Boss Room is server-authoritative by default" [7], which has been adopted by the first author who developed the latter.

Related Patterns. High-Level Branching, Authoritative Branching, and Low-Level Branching require a decision on whether to implement a networked behavior server- or client-authoritative. This pattern helps making such a decision.

4.2 High-Level Branching

Context. When implementing networked behaviors in a project dealing with a predominant server or client authority, you usually need to execute parts of your logic only on the client or the server.

Problem and Forces. How can you achieve coarse-grained control of whether code is executed every frame on the server or client? You want to implement server- or client-specific behavior, accepting to exclude the presence of both in the same class. You need coarse-granular control of where code is executed, working with a fixed server or client authority. You want to reduce the risk of executing code on the wrong machine. You want to avoid spaghetti code.

Solution. Disable the game loop on `Awake` and enable it only for the server (or client) instance when the NetworkBehavior gets initialized. Add the suffix "Server" for server-only and "Client" for client-only code to the class name to show where the code is executed.

Example. Consider the logic of computing paths on a navigation mesh. Let's assume you want to move characters along computed paths on the server and synchronize their positions with clients. Therefore, computing paths and further logic shall be executed on the server only.

```
public class NavMeshControlServer : NetworkBehaviour {
  // Awake is called before the first frame
  void Awake() {
    // Disable game loop
    enabled = false;
  }
```

```
  // OnNetworkSpawn is called by the NetworkBehaviour for
      initialization
  public override void OnNetworkSpawn() {
    base.OnNetworkSpawn();

    // Code in Update shall be executed on server only
    if (!IsServer)
      return;

    // Enable game loop only for server instances
    enabled = true;
  }

  // Update is called every frame but only on the server
  void Update() {
    // Logic to compute the navigation mesh path
    var navMeshPath = ComputeNavMeshPath();
    // Assume we have a navMeshAgent to apply the path to
    navMeshAgent.SetPath(navMeshPath);
    // Continue here with further logic, e.g., moving the
        agent
    [...]
  }
}
```

Consequences. This pattern has the following benefits: (B1) You have a per-class coarse-grained control over where code is exclusively executed. (B2) It can be derived from the class name where code is being executed. (B3) Splitting server-only and client-only code by classes prevents code tangling and encourages to think about architecture. The pattern's liabilities are as follows: (L1) You cannot mix server- and client-only code in the same class and, therefore, do not have fine-grained control. (L2) For simple behaviors, splitting server- and client-only code into separate classes makes their interaction unnecessarily complex. (L3) When there is a need to change authority, one line of each class needs to be changed.

Known Uses. This pattern can be found in two games. It occurs 30 times in "Boss Room", for example here, and 9 times in The Shroomlings, for example here.

Related Patterns. Low-Level Branching and Authoritative Branching also deal with questions about where code should be executed, focusing on fine-grained control. However, when the entire logic of a class is intended to be either server- or client-authoritative, High-Level Branching enforces a clear separation. Server First helps decide on whether to apply server- or client-authority.

4.3 Low-Level Branching

Context. When implementing networked behaviors in a project dealing with a predominant server or client authority, you usually need to execute parts of the game logic only on the client or the server to achieve consistent behavior across clients.

Problem and Forces. How can you achieve fine-grained control of whether the code is executed on the server or the client? It is important to implement server- or client-specific behavior without excluding the presence of both in the same class.

Solution. Wrap code with an if-block using NetworkBehaviour's `IsServer` for code being executed on the server and `IsClient` for code being executed on the client, respectively.

Example. Consider the logic of updating a client's graphical user interface (GUI) because the player got a level up. The handling of GUI elements is client-specific and does not exist in a server build. Therefore, we can handle the GUI update from the client's side.

```
if (IsClient) {
  guiHandler.HandleNewLevel(newLevel);
}
```

Consequences. This pattern has the following benefits: (B1) You have fine-grained control over where code is exclusively executed. (B2) You can immediately see whether the wrapped code is executed on the server or the client. (B3) You can mix server- and client-only code in the same class, making their interaction visible in one place. The pattern's liabilities are as follows: (L1) Applying the pattern multiple times in the same class leads to code tangling. (L2) It makes understanding the overall server or client context for complex cases hard. (L3) When there is a need to change authority, each if statement needs to be rewritten.

Known Uses. This pattern can be found in two games. It occurs 75 times in Boss Room (11x `IsClient`, 1x `!IsClient`, 44x `IsServer`, and 19x `!IsServer`), for example here, and 23 times in The Shroomlings (3x `IsClient`, 7x `IsServer`, and 13x `!IsServer`), for example here.

Related Patterns. A pattern that extends this one is Authoritative Branching. High-Level Branching also deals with questions about where code should be executed, and should be preferred when the entire logic of a class is intended to be either server- or client-authoritative. Server First helps decide on whether to apply server- or client-authority.

4.4 Authoritative Branching

Context. When implementing networked behaviors in a project dealing with server- or client-authoritative modes, you usually need to execute parts of your logic only on the authoritative or non-authoritative side.

Problem and Forces. How can you achieve fine-grained control of whether code is executed on the server or client while being flexible when changing the authoritative mode? You want to implement authoritative or non-authoritative behavior without excluding the presence of both in the same class. You want to change the authoritative mode during development. You need fine-granular control of where code is executed while having the option to switch between server- or client-authoritative behavior easily.

Solution. Wrap code with an if-block using a condition `HasAuthority` to execute code on the authoritative side. The condition `HasAuthority` can be set per class or globally, using NetworkBehavior's `IsServer` or `IsClient`.

Example. Consider the logic of moving a character and synchronizing its position. You might start with implementing the logic server-authoritative to benefit from security and consistency. However, during testing, the initial server-authoritative mode leads to unacceptable response times for the character to move on clients. Therefore, you want to switch from server-authoritative to client-authoritative. Thanks to Authoritative Branching, you can easily switch back and forth.

```
public class SomeClass : NetworkBehaviour {
  // Switch to IsClient for client-authoritative
  bool HasAuthority => IsServer;

  // Update is executed every frame
  void Update() {
    if (HasAuthority) {
      var newPos = GetNewPosition();
      // This method uses the Netcode for Gameobjects API to
          synchronize the position across the network
      UpdateSynchronizedPosition(newPos);
    }
    // For both having and not having authority, apply the
        synchronized position
    transform.position = GetSynchronizedPosition();
  }
}
```

Consequences. This pattern has the following benefits: (B1) You can easily change between server or client-authoritative in the context of a class or even globally. (B2) You have fine-grained control over where code is exclusively executed. (B3) You can mix authoritative and non-authoritative code in the same class, making their interaction visible in one place. The pattern's liabilities are

as follows: (L1) To know whether code runs on the client or server, you must look at how `HasAuthority` is defined. (L2) Applying the pattern multiple times in the same class leads to code tangling. (L3) It makes understanding the overall authoritative or non-authoritative context for complex cases hard.

Known Uses. This pattern can be found in one game. It occurs once in Boss Room. However, the development team documented this pattern [7]. The Shroomlings does not use this pattern but applies the patterns High-Level Branching and Low-Level Branching instead.

Related Patterns. This pattern extends the pattern of Low-Level Branching by extracting the `IsServer` or `IsClient` calls to a `HasAuthority` condition for improved flexibility. High-Level Branching also deals with questions about where code should be executed, and should be preferred when the entire logic of a class is intended to be either server- or client-authoritative. Server First helps decide on whether to apply server- or client-authority.

4.5 Continuous Synchronization

Context. When game state variables are changing, you need to think about how to keep them synchronized across clients.

Problem and Forces. How can you synchronize variable changes continuously and deliver the latest state to late-joining clients? You want to achieve consistency by distributing state changes to all clients. You want late-joining clients to receive the latest state.

Solution. Use `NetworkVariable<T>` to synchronize your type `T` continuously. Due changes will be distributed on every tick, and the current state will be distributed to newly joining clients.

Example. Consider the health points of units. They shall be synchronized with every state change so that all clients agree on consistent health points, and players will see synchronized health bars.

```
// Declare the network variable
private readonly NetworkVariable<uint>
_currentHealthPoints = new();

// Register to state change events
_currentHealthPoints.OnValueChanged += (oldValue, newValue)
    => {
  // You have a healthBar object, which can update its
      visuals according to an unsigned integer
  healthBar.Update(newValue);
}
```

Consequences. This pattern has the following benefits: (B1) Easy-to-use, out-of-the-box solution with low development effort. (B2) Late-joining clients get the

latest state of the game. The pattern's liabilities are as follows: (L1) Consumes bandwidth for each state change, which might be unnecessary due to some tolerance. (L2) For complex types, it might be more efficient to implement a custom network variable, inheriting from `NetworkVariableBase`, instead of using the generic `NetworkVariable<T>`. (L3) Since the solution is easy to use and out-of-the-box, you might be tempted to synchronize state that does not need to be synchronized. Such a state could be deterministically derived from other already synchronized states.

Known Uses. This pattern can be found in two games. It occurs 22 times in Boss Room, for example here, and 7 times in The Shroomlings, for example here.

Related Patterns. One-Shot Synchronization also approaches achieving consistency. However, it does not account for late-joining clients. In essence, Continuous Synchronization is convenient if bandwidth resources are not an issue.

4.6 One-Shot Synchronization

Context. When game events are happening, you need to think about how to keep them synchronized across clients. Late-joining clients do not need to receive past events.

Problem and Forces. How can you synchronize events between connected clients and the server in both directions? You want to achieve consistency by distributing events from the server to clients or from a client to the server.

Solution. Use `ServerRpc` to synchronize from a client to the server and use `ClientRpc` to synchronize from the server to the clients.

Example. Consider a player who wants to resign from the game. The event of the player's resignation must be synchronized with the server, and the consequences must be synchronized with the remaining clients.

```
// Sent from a client to the server
[ServerRpc]
private void ResignServerRpc(
  ServerRpcParams serverRpcParams = default
) {
    // Executed on the server
    HanldeClientResign(serverRpcParams.Receive.SenderClientId);
}
```

Consequences. This pattern has the following benefits: (B1) Easy-to-use, out-of-the-box remote procedure calls (RPC). (B2) RPC calls are visible thanks to the suffix of the forced method name. (B3) Newly joining clients do not get past RPCs, which saves bandwidth. The pattern's liabilities are as follows: (L1) Client-to-client remote procedure calls are not possible with ServerRpc or ClientRpc. Note that since NGO version 1.8.0, there is a new Rpc attribute for

that purpose. (L2) Newly joining clients do not get past RPCs, which needs careful treatment. (L3) Since RPCs are easy to use and out-of-the-box, you might be tempted to synchronize events that do not need to be synchronized. Determinism might solve the underlying issue as well.

Known Uses. This pattern can be found in two games. It occurs 52 times in Boss Room (29x "ServerRpc" and 23x "ClientRpc"), for example here, and 9 times in The Shroomlings (8x "ServerRpc" and 1x "ClientRpc"), for example here.

Related Patterns. Continuous Synchronization also approaches consistency, but accounts for late-joining clients, which causes additional bandwidth costs. On the contrary, One-Shot Synchronization generally envisions an effective balance between synchronization needs and bandwidth resource consumption.

4.7 Sanity Check

Context. When you program game mechanics, ensuring a player cannot cheat is essential for most online multiplayer games.

Problem and Forces. How can you prevent players from cheating on the client side? You want to ensure that the game logic is such that users cannot cheat, even if the client build is modified for that purpose. Hacking the client build must not lead to world consistency issues and, therefore, must not interfere with the experience of non-cheating players.

Solution. Apply server authority when synchronizing state. In addition to validating a user action on the client side for responsiveness, the same validation must be used on the server side for consistency.

Example. Consider that a player can build defense towers on the terrain. There are two types of terrain: grass and water. The player cannot build towers on water. Therefore, you must validate that the creation of defense towers does not happen on water.

```
void ApplyUserInput(Input userInput) {
  // This if statement could be removed by a hacker to omit
      validation
  if (TerrainTypeValidator.IsNotOnWater(userInput)) {
    // Calling this method directly on the client does not
        work because the effects are server authoritative
    CreateDefenseTowerOnServerRpc(userInput);
  }
}

// Remote procedure call from the client to the server
[ServerRpc]
void CreateDefenseTowerOnServerRpc(userInput) {
```

```
  // Since this code runs on the server, it cannot be changed
     by a hacker not having access to the server applies the
     same validation as on client side; sanity check
  if (TerrainTypeValidator.IsNotOnWater(userInput)) {
    // the factory contains server-authoritative code only
    CreationFactory.makeDefenseTower(userInput);
  }
}
```

Consequences. This pattern has the following benefits: (B1) Hacking the client-side build does not affect world consistency across clients. (B2) The game logic is more resilient to cheating. The pattern's liabilities are as follows: (L1) State has to be synchronized server-authoritative. (L2) Responsiveness is more difficult to achieve. (L3) Validation is performed twice, which leads to additional load.

Known Uses. This pattern can be found in two games. It occurs 11 times in Boss Room, for example here, and once in The Shroomlings, here.

Related Patterns. This pattern is related to Server First, as both suggest to apply server authority.

4.8 Marshaling by Reference

Context. When you synchronize unpredictable but pre-configured events, it is essential to serialize those events.

Problem and Forces. How can you save bandwidth and serialization effort when communicating a configured complex event over the network? You want to save bandwidth when synchronizing an event while not restricting the complexity of the event itself. The event cannot be foreseen because it is triggered by, e.g., a user input. Further, you want to reduce the development effort spent on making the event serializable.

Solution. Ship a mapping of configured complex events by ID to the server and all clients. When communicating an event over the network, only synchronize the dynamic part of the event and use the referencing ID to the configured part instead of communicating the serialized form of the whole event.

Example. Consider a medieval shop where the trader offers dozens of items. When the player buys an item on the client, all the server needs to know is the ID for that item. It is not needed to serialize the whole item nor is it relevant how complex the item is.

```
// This map has the same configuration on server and clients
Map<ItemId, Item> itemsById = GetConfiguredItemsByID();

// The client calls a remote procedure on the server, and the
   id gets serialized
[ServerRpc]
void BuyItemServerRpc(ItemId id) {
```

```
  var item = itemsById.Get(id);
  // Add the item to the player's inventory
  player.AddToInventory(item);
}
```

Consequences. This pattern has the following benefits: (B1) Bandwith usage is reduced. (B2) CPU time spent for serialization is reduced. (B3) Development effort is reduced for serialization. The pattern's liabilities are as follows: (L1) If the server and clients do not have the same version, the configuration of objects might diverge. (L2) Cannot synchronize changes of instantiated objects, only static configuration. (L3) Need to handle the case where there's no object configured for a given ID.

Known Uses. This pattern can be found in two games. It occurs once in Boss Room, here, and once in The Shroomlings, here.

Related Patterns. Marshaling by Value also addresses the question of how to serialize objects, but serializes the whole object.

4.9 Marshaling by Value

Context. When you communicate objects over the network, these objects must be serializable.

Problem and Forces. How can you serialize when communicating an object over the network, even when you cannot change the object's definition? You want to serialize an object of a third-party library without changing that library code. You want to centralize the cross-cutting concern of serialization so you do not pollute your object definitions with serialization code.

Solution. Serialize all members of each object layer separately through custom serialization extensions. Put all serialization extensions in a dedicated namespace.

Example. Consider a system to distribute downloadable content (DLC). The server distributes a set of client-specific URLs so they can get the DLCs. When using a third-party class URL, which wraps a string, we have to provide serialization logic for that class so it can be communicated over the network.

```
// The definition of ForeignClass is handled by a third party
using ThirdParty;
// Use NativeArray from Unity.Collections to deal with lists
using Unity.Collections;
// We will use generic List<T> from
    System.Collections.Generic;
using System.Collections.Generic;

public class DLCDistribution {
  // Url is part of the assembly ThirdParty
  public List<Url> URLs;
```

```
}

namespace Serialization {
  // the class name does not matter
  public static class SerializationExtensions {
    // Serialization and deserialization of Url
    public static void ReadValueSafe(
      this FastBufferReader reader, out Url URL
    ) {
      // Deserialization
      reader.ReadValueSafe(out string val);
      url = new Url(val);
    }
    public static void WriteValueSafe(
      this FastBufferWriter writer, in Url URL
    ) {
      // Serialization
      writer.WriteValueSafe(url.Value);
    }

    // Serialization and deserialization of DLCDistribution.
       It is known how to serialize and deserialize Url
    public static void ReadValueSafe(
      this FastBufferReader reader,
      out DLCDistribution distribution
    ) {
      // Deserialization. NativeArray can be used to
         deserialize sets
      reader.ReadValueSafe(
        out NativeArray<Url> URLs, Allocator.Temp
      );
      distribution = new DLCDistribution(urls.ToList());
    }
    public static void WriteValueSafe(
      this FastBufferWriter writer,
      in DLCDistribution distribution
    ) {
      // Serialization
      writer.WriteValueSafe(distribution.urls.ToArray());
    }
  }
}
```

Consequences. This pattern has the following benefits: (B1) Works for serialization of 3rd party types. (B2) Centralizes the cross-cutting concern of serialization. (B3) Decouples serialization logic from type definitions. The pattern's liabilities are as follows: (L1) Not optimized for bandwidth. (L2) Not optimized for CPU. (L3) Unnecessary effort when serializing static information.

Known Uses. This pattern can be found in two games. It occurs once in Boss Room and once in The Shroomlings, here.

Related Patterns. Marshaling by Reference also addresses the serialization of objects. However, it does not serialize the whole object but instead uses references to the static part of a complex object.

5 Threats to Validity

Our pattern catalog has been developed based on only two real-time online multiplayer games. This means that the patterns might not be applied with the same success to other games as the ones analyzed. Careful comparisons for similarities with the characteristics of the analyzed games are crucial.

Moreover, our pattern catalog is not meant to be complete. It is likely that over time, new patterns will be documented, and existing ones might be further refined. This is especially true because the development of netcode libraries is a fast-developing field.

Finally, the "Known Uses" for each of the patterns might not be complete. Some occurrences might have been overlooked, as identifying and counting them accurately is inherently challenging when done semi-automatically.

6 Related Work

Game development has attracted considerable attention from researchers in the past decade, due to the various factors that differentiate it from software development in other domains [22,29]. Scholarly articles discussing the need for design patterns to advance game development date back almost 25 years [4,14,25], and more recent studies exploring the usage of design patterns in games can be found in [2,26]. However, these works focus on the usage of classical object-oriented patterns, notably the well-known GoF patterns [11]. As mentioned already in the introduction, this close connection to traditional design patterns is also evident in Nystrom's book [27], arguably the most influential practically oriented work on game development patterns to date. More recent research has approached the challenges of game development from a different angle, examining code smells and anti-patterns in game design, with the goal of helping developers avoid common pitfalls [5,1]. However, to the best of our knowledge, there is no existing work that specifically addresses the challenges of developing the networked aspects of online multiplayer games, neither in the form of patterns nor anti-patterns.

Patterns that are distantly related to ours can be found in the domain of distributed computing. To address the fundamental challenges of distributed systems in general, the proposed patterns range from bio-inspired patterns that mimic mechanisms found in biological systems [3]; over patterns for decentralized coordination in self-organizing, emergent systems [37,19]; up to security-focused patterns for distributed architectures [36]. Most of these patterns, however, operate at a much more fundamental level, tackling core problems of distributed

systems mostly from a conceptual point of view rather than through concrete design or implementation aspects. In contrast, our work is concerned with the design decisions developers face when implementing online multiplayer games using game engines that already encapsulate many of these fundamental concerns. A more practical, though largely domain-agnostic, treatment of design patterns for distributed systems was recently published by Unmesh Joshi [17]. Like our patterns, the thirty patterns presented in the book, an overview of which may be found on Martin Fowler's Blog[3], are accompanied by code examples to clarify essential implementation details. Although none of them specifically addresses game development, some of them show similarities to our patterns. For example, the patterns "Replicated Log" and "State Watch" address the general problem of maintaining consistent, shared state across nodes. This reappears in a specific form when the state of the game is synchronized between players, as addressed by our patterns Continuous Synchronization and One-Shot Synchronization. Moreover, the "Leader and Followers" pattern suggests using a single server to coordinate replication across a set of servers. This strategy is similar to the high-level idea behind our Server First pattern, although finer-grained authoritative control can be implemented using the patterns High-Level Branching, Low-Level Branching, and Authoritative Branching.

The demanding requirements of real-time simulation lead to Entity-Component System architectural patterns (ECS), which emphasize modularity, data orientation, and separation of concerns by composing behavior and data through entities, components, and systems. The recent book by Richard Johnson [16] presents a thorough reference for such concepts. Unity provides a library called "Netcode for Entities" that implements the ECS architecture. However, as mentioned in the introduction, the patterns in this work target the "Netcode for GameObjects" library, applying conventional object-oriented programming paradigms.

7 Conclusion

In this paper, we presented a first catalog of netcode patterns using Unity's Netcode for GameObjects library, compiling documented solutions for recurring challenges when developing real-time online multiplayer games. We used two concrete games of this genre that serve as case studies for pattern identification and validation. Thereby, we also contribute a game artifact, The Shroomlings, a competitive Real-Time Strategy game whose ongoing development will contribute to the validation of the pattern catalog and may reveal additional limitations or new patterns in the future. To further validate and develop the patterns' generality and make our catalog accessible to a broader audience and additional (open-source) games, our patterns could be transferred to other game engines such as Unreal [10] or Godot [18].

[3] https://martinfowler.com/books/patterns-distributed.html

8 Acknowledgments

We are very grateful to Dominik Deimel for shepherding this paper for EuroPLoP 2025. His high-quality feedback has been a great support in shaping this paper.

References

1. Agrahari, V., Shanbhag, S., Chimalakonda, S., Rao, A.E.: A catalogue of game-specific anti-patterns based on github and game development stack exchange. Journal of Systems and Software **204**, 111789 (2023)
2. Ampatzoglou, A., Chatzigeorgiou, A.: Evaluation of object-oriented design patterns in game development. Information and Software Technology **49**(5), 445–454 (2007)
3. Babaoglu, O., Canright, G., Deutsch, A., Caro, G.A.D., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., et al.: Design patterns from biology for distributed computing. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **1**(1), 26–66 (2006)
4. Björk, S., Holopainen, J.: Games and design patterns. The game design reader: A rules of play anthology pp. 410–437 (2005)
5. Borrelli, A., Nardone, V., Di Lucca, G.A., Canfora, G., Di Penta, M.: Detecting video game-specific bad smells in unity projects. In: Proceedings of the 17th international conference on mining software repositories. pp. 198–208 (2020)
6. Casamayor, R., Arcega, L., Pérez, F., Cetina, C.: Bug localization in game software engineering: evolving simulations to locate bugs in software models of video games. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems. pp. 356–366 (2022)
7. Draper, K.: Boss room's authority (2024), https://docs-multiplayer.unity3d.com/netcode/current/learn/dealing-with-latency/#boss-rooms-authority
8. EuroPLoP: Patterns (2024), https://www.europlop.net/patterns/
9. Feldmeier, P., Fraser, G.: Combining neuroevolution with the search for novelty to improve the generation of test inputs for games. In: Proceedings of the 1st ACM International Workshop on Foundations of Applied Software Engineering for Games. pp. 14–19 (2024)
10. Games, E.: Unreal engine (2025), https://www.unrealengine.com/
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Pearson (1995)
12. Haas, J.K.: A history of the unity game engine (2014), https://api.semanticscholar.org/CorpusID:86824974
13. Harrison, N.B.: Advanced pattern writing (2004)
14. Holopainen, J., Björk, S.: Game design patterns. Lecture Notes for GDC (2003)
15. House, B.: How to choose the right netcode for your game (2023), https://blog.unity.com/games/how-to-choose-the-right-netcode-for-your-game
16. Johnson, R.: Entity-Component System Design Patterns: Definitive Reference for Developers and Engineers. HiTeX Press (2025)
17. Joshi, U.: Patterns of distributed systems. Addison-Wesley Professional (2023)
18. Juan Linietsky, A.M.: Godot engine (2025), https://godotengine.org/
19. Juziuk, J., Weyns, D., Holvoet, T.: Design patterns for multi-agent systems: A systematic literature review. In: Agent-Oriented Software Engineering. pp. 79–99. Springer (2014)

20. Liu, S., Xu, X., Claypool, M.: A survey and taxonomy of latency compensation techniques for network computer games. ACM Computing Surveys (CSUR) **54**(11s), 1–34 (2022)
21. Metzger, F., Geißler, S., Grigorjew, A., Loh, F., Moldovan, C., Seufert, M., Hoßfeld, T.: An introduction to online video game qos and qoe influencing factors. IEEE Communications Surveys & Tutorials **24**(3), 1894–1925 (2022)
22. Murphy-Hill, E., Zimmermann, T., Nagappan, N.: Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development? In: Proceedings of the 36th international conference on software engineering. pp. 1–11 (2014)
23. Mzid, R., Rezgui, I., Ziadi, T.: Attention-based method for design pattern detection. In: European Conference on Software Architecture. pp. 86–101. Springer (2024)
24. Newzoo: Global games market report (2024), https://newzoo.com/resources/trend-reports/newzoos-global-games-market-report-2024-free-version
25. Nguyen, D., Wong, S.B.: Design patterns for games. In: Proceedings of the 33rd SIGCSE technical symposium on Computer science education. pp. 126–130 (2002)
26. Nikolaeva, D., Safi, M., Mihailov, M., Georgiev, A., Bozhikova, V., Stoeva, M.: Algorithm a* and design patterns used in unity video game development. In: 2020 International Conference Automatics and Informatics (ICAI). pp. 1–3. IEEE (2020)
27. Nystrom, R.: Game programming patterns. Genever Benning (2014)
28. Paduraru, C., Stefanescu, A., Jianu, A.: Unit test generation using large language models for unity game development. In: Proceedings of the 1st ACM International Workshop on Foundations of Applied Software Engineering for Games. pp. 7–13 (2024)
29. Pascarella, L., Palomba, F., Di Penta, M., Bacchelli, A.: How is video game development different from software development in open source? In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 392–402 (2018)
30. Politowski, C., Petrillo, F., Guéhéneuc, Y.G.: A survey of video game testing. In: 2021 IEEE/ACM International Conference on Automation of Software Test (AST). pp. 90–99. IEEE (2021)
31. Reeve, A.: Networking components (2024), https://docs-multiplayer.unity3d.com/netcode/current/basics/networkobject/
32. Technologies, U.: Script lifecycle flowchart (2024), https://docs.unity3d.com/Manual/ExecutionOrder.html
33. Tim Wellhausen, A.F.: How to write a pattern? (2011)
34. Unity: Enter the boss room (2024), https://unity.com/demos/small-scale-coop-sample
35. Unity: Getting started with boss room (2024), https://docs-multiplayer.unity3d.com/netcode/current/learn/bossroom/bossroom
36. Uzunov, A.V., Fernandez, E.B., Falkner, K.: Securing distributed systems using patterns: A survey. Computers & Security **31**(5), 681–703 (2012)
37. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M.: On patterns for decentralized control in self-adaptive systems. In: Software engineering for self-adaptive systems II: international seminar, dagstuhl castle, Germany, october 24-29, 2010 revised selected and invited papers. pp. 76–107. Springer (2013)