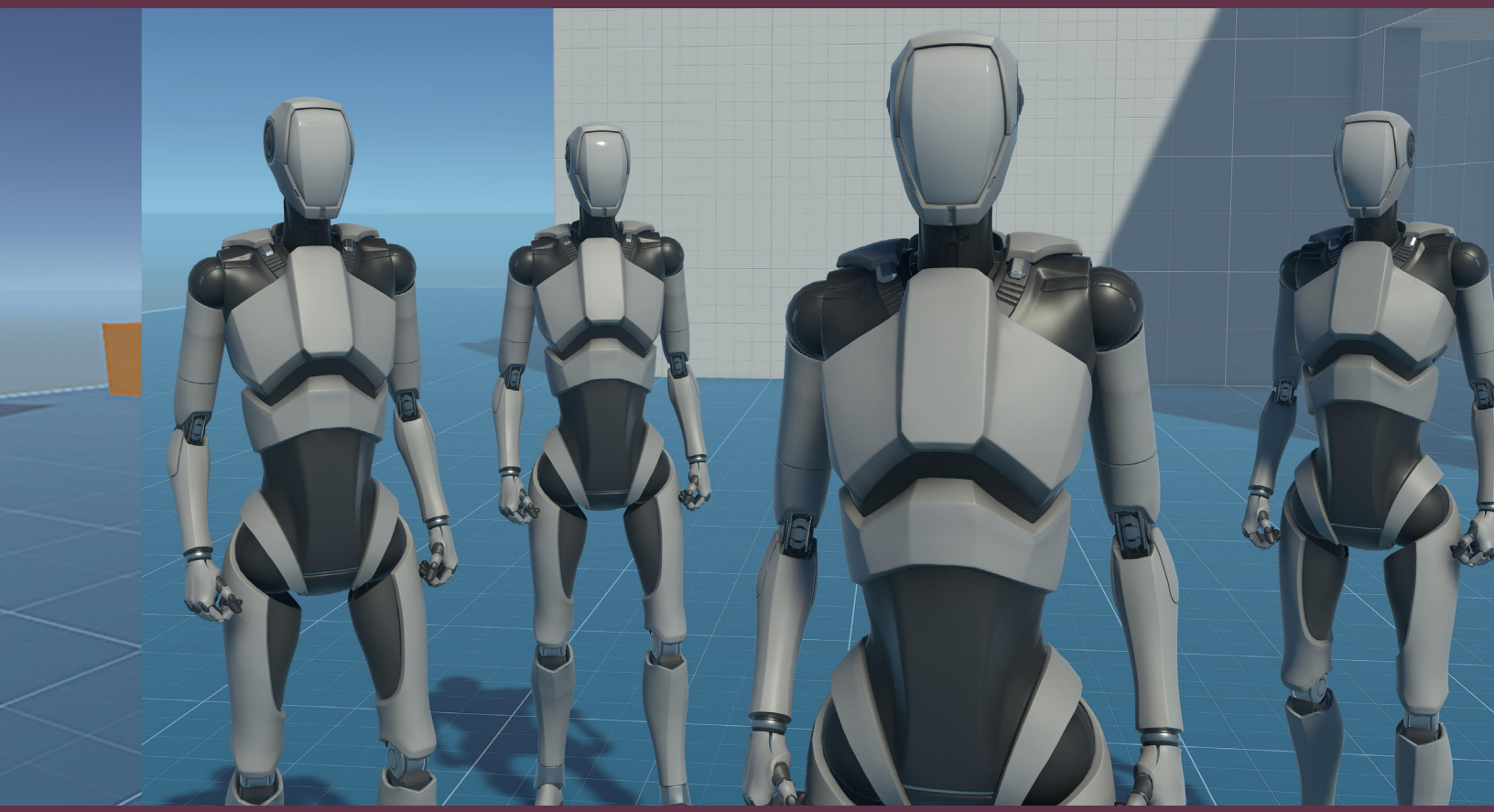
→ EBOOK

Unity におけるマルチプレイヤーネットワーク入門



Unity®

Contents

はじめに

オンラインゲームで人々が集まると、特別な何かが生まれます。シンプルなレーシングゲームや RPG も、単なるゲームから共有体験へと変わります。そこで起こることは予測不能で、挑戦しがいがあり、そして何よりも楽しいものです。

宇宙の侵略者と戦うためにチームを組んだり、シューティングゲームで対戦したりする場合、ネットワークを介したマルチプレイヤーゲームでは、物理的な境界を越えて協力、戦闘することができます。この集団的なインタラクションこそが、マルチプレイヤー体験の本質です。

しかし、ネットワークを介したゲームプレイの開発は、同等のシングルプレイヤーアプリケーションの開発よりも複雑になる場合があります。

本ガイドの目的は、Unity のネットワークツールを使用したマルチプレイヤーゲームの開発を始めるユーザーをサポートすることです。ネットワークの概念に関する基礎知識を紹介し、Unity のネットワークサンプルを試す前の入門ガイドとしての役割を果たします。また、Unity Asset Store で入手できる Starter Assets – ThirdPerson パッケージの基本的なユースケースについて説明しています。

本ガイドは、Unity と C# の開発には精通しているものの、ネットワーク開発は未経験または初心者のユーザーを対象としています。マルチプレイヤー開発の理論的な側面をすばやく理解し、実践的なデモに向けて準備できるように構成されています。

この eBook では以下の内容を取り上げます。

— Unity のマルチプレイヤーの基本概念について学ぶ

— さまざまなマルチプレイヤーシステムとネットワーキングモデルについて学ぶ

— Netcode for GameObjects を使用した簡単な例を設定する

Unity におけるマルチプレイヤーゲーム向けツールの進化

Unity は数多くのマルチプレイヤーゲーム開発ツールおよびソリューションを提供しています。これには、Netcode for GameObjects や Netcode for Entities のフレームワークに加え、Game Server Hosting (Multiplay) やボイスチャットおよびテキストチャット機能を備えた Vivox などが利用できる Unity Gaming Services (UGS) が含まれます。カジュアルな協力型ゲームでも、オープンワールドの MMO でも、Unity を使用すればマルチプレイヤー開発をスムーズに始めることができます。

Unity 6 には、マルチプレイヤーゲーム向けの新機能と改善された機能が搭載されており、統合、イテレーション、デプロイの信頼性と速度がこれまで以上に向上しました。Unity 6 のマルチプレイヤーゲーム向けの新機能を簡単に紹介します。

- **Multiplayer Center**:Unity 6 のコアパッケージとして入手可能な Multiplayer Center は、マルチプレイヤーゲームの設定と開発をより簡単にします。新しいプロンプトとワークフローでは、プロジェクトの開始に使用できる動的テンプレートを生成する前に、ゲームのパラメーターと要件に基づいて、関連するパッケージやサービスを提案します。
- **Multiplayer ウィジェット** :Multiplayer ウィジェットは、Unity Gaming Services (UGS) をマルチプレイヤーゲームに統合するのに役立ちます。スタンドアロンのパッケージとして利用することも、Multiplayer Center からアクセスすることも可能です。
- **Multiplayer Tools パッケージ** : このパッケージは、Unity におけるマルチプレイヤーゲームの開発ワークフローを改善し、Netcode for GameObjects 2.0 によってパフォーマンスを向上させ、Distributed Authority に対応できるようにします。
- **Multiplayer Play Mode**: このパッケージは、ディスク上の同じアセットから最大 4 つの独立した軽量エディタープロセスを起動することで、ゲームプレイの検証プロセスを効率化します。非常に野心的なサーバーホスト型のプロジェクトでは、Play Mode Scenarios を活用することで、専用サーバーのビルドや Multiplay Hosting サーバーへの直接アップロードなどのデプロイのステップを設定できます。

— **Distributed Authority (ベータ版)**:2024 年 11 月時点で、Distributed Authority のベータ版は、Netcode for GameObjects と連携して使用できます。これにより、ゲームセッション中にスポーンされた NetcodeObject の権限をクライアント間で分散所有できるようになります。ネットコードシミュレーションのワークロードをクライアント間で分散しつつ、Unity が提供する高パフォーマンスのクラウドバックエンドを通じてネットワークステートを一元管理します。

— **Multiplayer Services SDK**: この SDK は、Unity 6 で開発されたゲームにマルチプレイヤー要素を加えるためのワンストップソリューションです。Relay や Matchmaker といった、Unity Gaming Services (UGS) の Multiplayer サービスは、プレイヤーのグループがゲーム内でどのように相互作用するかを定義するために、Sessions を活用します。Sessions は、ネットワーキングライブラリの Netcode for GameObjects または Netcode for Entities のどちらでもでも動作します。

Unity 6 のマルチプレイヤーソリューションの詳細については、以下のリソースを確認してください。

— 【日本語吹き替え】競争力のあるマルチプレイヤーゲームの開発を加速させる｜Unite 2024

— 【YouTube 自動翻訳字幕推奨】マルチプレイヤー対応: スタジオとゲームを成功に導く方法

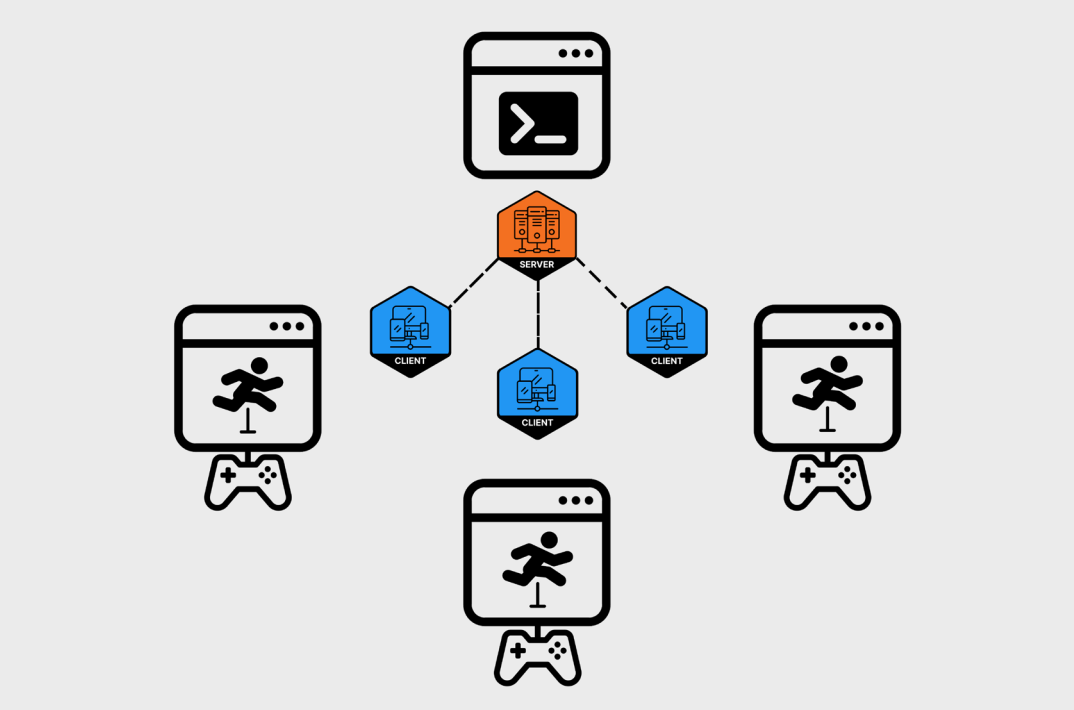
— Unity マニュアル：Unity 6 における Multiplayer の新機能

— Unity 6、登場。新機能をご覧ください

Unity の一連のツールは、コンセプトからプロトタイピング、リリース、継続的な運用まで、マルチプレイヤーゲームの開発ライフサイクル全体をサポートします。Unity エコシステム内で完結することも、チームのニーズ応じて最適なツールやサービスを選択することもできます。

基本概念

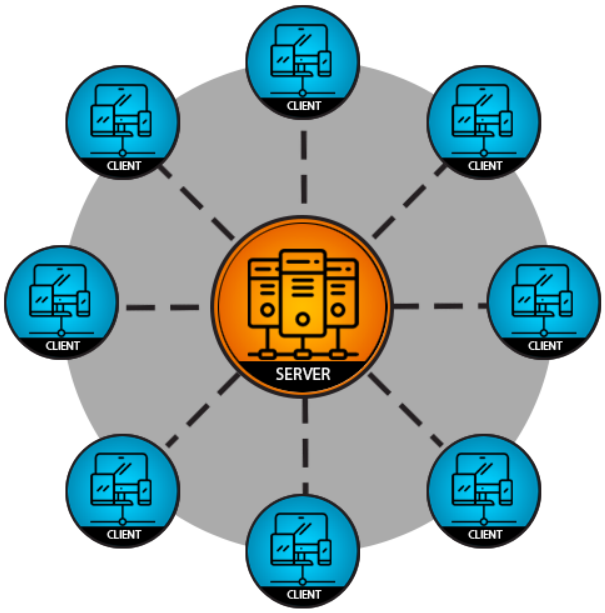
マルチプレイヤーゲームでは、ネットワーキングによって、プレイヤーが中央のサーバーに接続したり、プレイヤー同士が互いに直接接続したりできます。これにより、リアルタイムでデータを共有して一緒にプレイすることが可能になります。同じゲームアプリケーションが各プレイヤーのデバイスで同時に実行され、各プレイヤーのアクションがネットワークを通じて同期されます。



ネットワークを介したマルチプレイヤーゲームでは、同じゲームアプリケーションが複数のデバイスで実行されます。

ネットワークを介した一般的なゲームのアーキテクチャは、**クライアント**と**サーバー**の 2 つの主要コンポーネントで構成されます。クライアントとは、プレイヤーのデバイス（PC、コンソール、スマートフォン）で動作するゲームのインスタンスを指します。クライアントは、ゲームグラフィックスのレンダリング、オーディオ再生、ユーザー入力のハンドルを行うほか、プレイヤーのアクションに関する更新情報をサーバーに送信します。

一方、サーバーはゲームのステート（ゲーム内における全要素の現在の状態）を管理します。クライアント間の通信をハンドルし、ゲームのルールを適用します。サーバーには、ゲーム開発者によって実行される専用のヘッドレスマシンと、プレイヤーの 1 人がサーバーの役割を担うプレイヤーホスト型の 2 種類があります。



クライアントサーバー型のアーキテクチャ

サーバーはクライアントからの入力を受け取り、ゲームロジックを処理して、クライアントに更新情報を返します。ゲームステートの最終的な権限はサーバーが持ちますが、応答性の高いゲームプレイを実現するために、クライアントでもローカルコピーを保持します。継続的な同期により、すべてのプレイヤーにとって一貫性のあるゲームプレイがリアルタイムで維持されます。

ヘッドレスサーバー

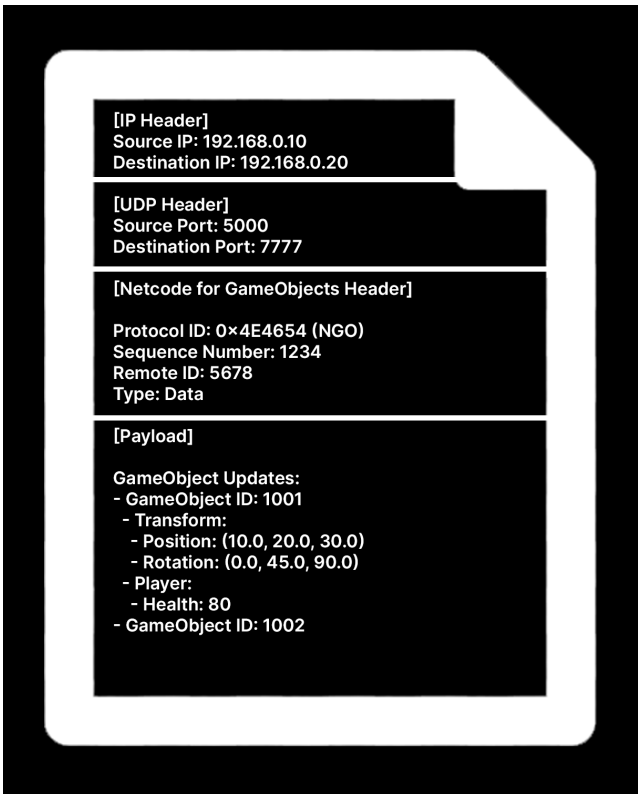
ヘッドレスサーバーとは、グラフィカルユーザーインターフェース（GUI）を持たず、バックエンドタスクにのみ特化して動作するサーバーを指します。

グラフィックスをレンダリングしないため、より簡単にスケールでき、多くの場合、専用ハードウェアやクラウド環境にデプロイされます。詳細については、「**ネットワークトポロジー**」の「**専用ゲームサーバー**」を参照してください。

UDP パケット

クライアントとサーバーは、UDP（ユーザーデータグラムプロトコル）などの標準的なインターネットプロトコルを使用してデータパケットを交換することで、通信を行います。リアルタイムアプリケーション、特に一人称シューティングゲームのようなテンポの速いゲームでは、TCP（伝送制御プロトコル）よりもUDP が好まれます。UDP を使用することで、ゲームが通信のさまざまな側面の優先順位を完全に制御できるためです。TCP とは異なり、UDP は受信者からの確認応答を必要としません。これにより、UDP はより効率的で柔軟な基盤となりますが、こうした機能が必要とされる場合、機能をハンドルする負担はアプリケーションにかかります。

各 UDP パケットは、**ヘッダーとペイロードで**構成されます。UTP ペイロードには、さらにプロトコル固有のセクションがあり、それぞれが独自のヘッダーとペイロードを持ちます。ネットワーク用語では、このようなネスト構造を**カプセル化**と呼びます。



UDP パケットの簡略版

IP ヘッダーと UDP ヘッダーはサイズが固定されており、送信者と受信者のアドレス（IP アドレスとポート）などの重要なメタデータが含まれます。ペイロードのサイズ、構造体、コンテンツは、特定のゲームやコンテキストによって異なります。例えば、ペイロードはプレイヤー入力や、特定の瞬間のゲームステートのスナップショットを含むことがあります。

UDP パケットはリアルタイムアプリケーションに不可欠な短い待ち時間と高速通信に適していますが、このスピード特性のトレードオフとして信頼性が欠けます。UDP プロトコルには、信頼性、順序制御、輻輳制御のメカニズムがありません。インターネット上で送受信されるパケットには、以下のようなエラーが発生する可能性があります。

— **パケットロス：**パケットが失われ、宛先に届かない場合があります。原因としては、ネットワークの輻輳やハードウェアの障害などの問題が考えられます。

— **重複 :** パケットが重複し、同じパケットが受信者に複数回届く場合があります。これは、ネットワークハードウェアまたはソフトウェアの設定ミスが原因で発生する可能性があります。

— **順序の乱れ :** パケットが、送信された順序とは異なる順序で受信者に届く場合があります。これは、パケットが宛先に届く際に異なるルートを通り、それぞれのルートで待ち時間が異なる場合に発生する可能性があります。

— **破損 :** パケットのコンテンツが伝送中に変更され、データが使用できなくなる場合があります。

UDP と TCP の比較

ネットワークプロトコルとは、ネットワーク上でデータを送受信する方法を制御する一連のルールおよび規則です。プロトコルは、接続の確立方法、メッセージの形式、エラーのハンドル方法、データの伝送方法を定義します。

通常、ゲーム開発では伝送制御プロトコル (TCP) よりも UDP が好まれます。これは、UDP が高速かつ軽量であるためです。TCP は信頼性が高く、ウェブブラウジングに適しています。失われたパケットや順序が乱れたパケットを再送信することで、正しい順序のデータ配信を保証します。しかし、ラグやスタッターが発生する可能性があるため、リアルタイムのゲームには適していません。

一方、UDP はリアルタイムのパフォーマンスを優先するために、一定のデータ損失を許容します。そのため、重要でないデータが一部ドロップした場合でも、ゲームはフリーズすることなく、60 FPS 以上で滑らかに実行できます。UDP は通常、応答性と時折のデータ損失のバランスが取れるため、ネットワークを介したマルチプレイヤーゲームに最適です。

UDP の信頼性の低さを補うために、以下のような手法が活用できます。これらは TCP のビルトイン機能とは異なり、リアルタイムゲームで使用できるように細かく調整できます。

手法	機能
シーケンス番号	各パケットには、一意かつ増え続けるシーケンス番号が付与されます。受信側はこれを使用して、パケットの欠落や順序の誤りを検出します。
確認応答 (ACK) と ACK ビットマスク	パケットには最後に受信したパケットのシーケンス番号が含まれており、送信側はどのパケットが届いたのかを把握できます。ACK ビットマスクは複数のパケットの状態を一度に追跡するため、パケットロスをより迅速に検出できます。
再送信タイムアウト調整 (RTO)	これは、TCP の仕組みを参考にしてラウンドトリップタイムを測定する手法です。例えば、ラウンドトリップタイムを測定し、それを使用してクライアント側の補間や予測を調整できます。
タイムアウト	一定時間内に確認応答を受信しない場合、パケットは失われたと見なされます。

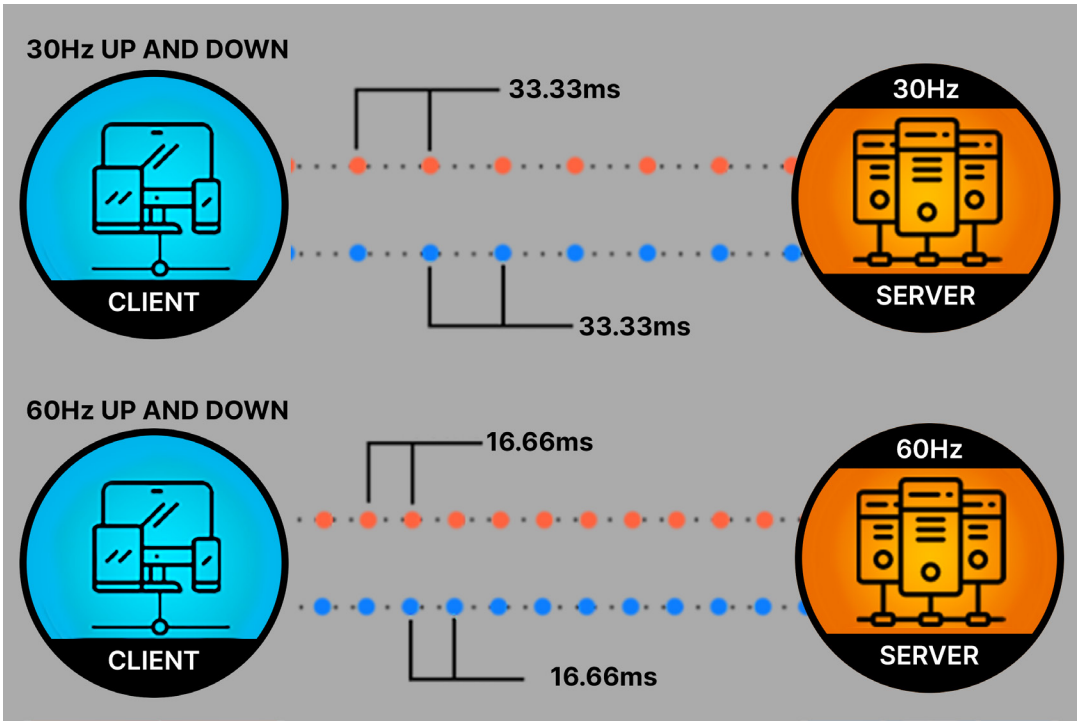
ティックと更新

マルチプレイヤーゲームでは、サーバーにディスプレイがなくても、コアとなるゲームロジック、物理演算シミュレーション、その他のゲームプレイ機能をハンドルします。

サーバーは、グローバルなゲームステートのみをハンドルしており、入力はクライアント側から受け取ります。これは、シングルプレイヤーゲームがローカルプレイヤーからの入力をハンドルするのとよく似ています。マウスやキーボードを使用する代わりに、サーバーはクライアントから送られていくる入力を処理することで、“権威あるゲームステート “を維持します。これには、プレイヤーの現在位置やオブジェクトステートから、物理演算、ゲームの進捗まで、あらゆるものが含まれます。

サーバー側の処理で要となるのが**サーバーティック**です。サーバーティックとは、受信された入力に基づいてサーバーがゲームステートを更新するサイクルを指します。これは、ティックレートと呼ばれる一定の間隔で発生し、ヘルツ (Hz) または 1 秒あたりのティック数で測定されます。このティックレートによって、ゲームワールドの更新頻度が決まります。

一方、**更新レート**とは、クライアントがサーバーとデータを交換する頻度を指します。更新頻度を上げると、ゲームの応答性を強化できますが、帯域幅と処理能力を多く消費します。これは通常、クライアントのネットワーク環境やコンピューティングリソースによって制約を受けます。



ティックレートと更新レートの比較

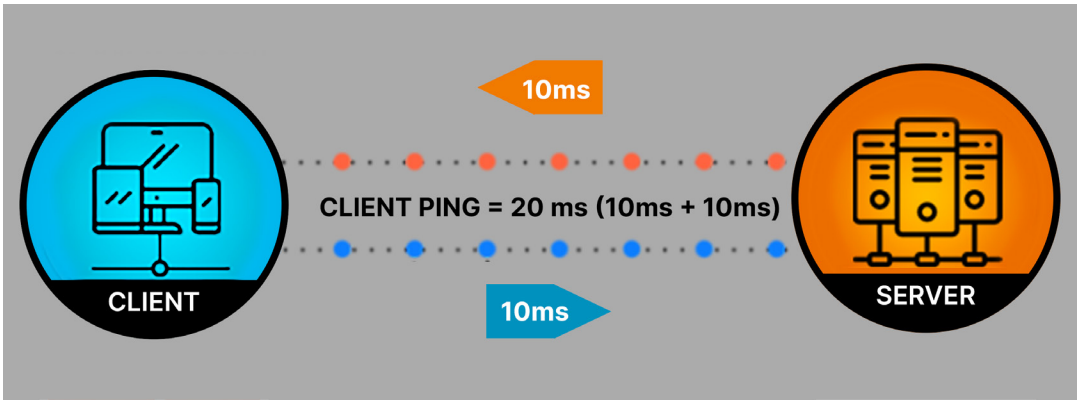
ティックレートを高くすると、ゲームの応答性は向上しますが、サーバーリソースに負荷がかかる可能性があります。同様に、更新頻度を上げるとインタラクションの滑らかさが増しますが、データ伝送量が増加します。滑らかで応答性の高いマルチプレイヤー体験を実現するには、ティックレートと更新レートの適切なバランスを見つけることが重要です。

実際のティックレートは、ゲームプレイのニーズによって異なります。テンポの速い一人称シューティングゲームでは、プレイヤーのすばやい動きや瞬時の射撃を反映するために、多くの場合 60 Hz 以上のティックレートで実行されます。一方、リアルタイムストラテジーゲームでは、素早い反応を必要としないため、ティックレートは 30 Hz で十分です。一方、大規模のストラテジー MMO では、膨大な数の同時プレイヤーをサポートするために、10Hz というかなり低いティックレートが使用されることがあります。

待ち時間

待ち時間とは、データが送信元から宛先に届くまでにかかる時間を指します。**ラウンドトリップタイム (RTT)** は、パケットが宛先に到達し、応答が返ってくるまでの時間を測定するもので、ネットワークの待ち時間を測る指標の 1 つとなります。

個人差はあるものの、一般的な目安として、待ち時間が約 200 ミリ秒になると、ユーザーはゲームプレイの品質低下に気づきます。ゲームの種類によって、許容できる待ち時間の程度は異なります。例えば、一人称シューティングゲームでは待ち時間が 100 ミリ秒未満の場合に最も優れたパフォーマンスを発揮し、リアルタイムストラテジーゲームでは待ち時間が最大 500 ミリ秒まで許容されます。



ラウンドトリップタイムはネットワーク待ち時間を測る 1 つの指標です。

待ち時間が短いと、プレイヤーの応答性が向上します。ユーザーのアクションから、マルチプレイヤーゲームで期待される結果が表示されるまでの遅延が最小限に抑えられていることが理想です。待ち時間が大きいと、ゲームプレイ中の遅延が目立つようになります。

ネットワーク以外の要因が原因となり、待ち時間が発生することもあります。例えば、ユーザー入力を検出する際に遅延が発生したり、レンダーパイプラインで一時的な処理の遅延が発生したりする場合があります。もう 1 つの原因は Vsync (垂直同期) です。これは画面のティアリングを防ぐ機能ですが、その代償として、さらなる待ち時間が生じます。

しかし、多くの場合、ネットワーク自体が待ち時間の主な原因です。これにはいくつかの遅延タイプが関与しています。

— **処理遅延 :** ルーターがパケットヘッダーを読み取り、パケットを宛先に転送する際にかかる時間です。通常は最小限ですが、この遅延は複数のルーターを経由することで積み重なり、影響が大きくなる場合があります。

— **伝送遅延:**これはパケットをネットワークに送信する際に要する時間で、パケットサイズから直接影響を受けます。帯域幅が狭いエンドユーザーネットワークではより顕著になります。

— **キューイング遅延:**輻輳やインターフェース容量の制限により、パケットがキューに滞留することで発生します。この遅延により、待ち時間が大幅に増加する可能性があります。

— **伝播遅延:**信号がネットワーク上を移動するのにかかる時間です。このタイプの遅延は主に、サーバーとユーザー間の物理的な距離、使用する物理メディア (光ファイバー、銅線、無線通信)、信号の種類 (電気、光、電波波形) が原因で発生します。

特にインターネットベースのゲームでは、ある程度の待ち時間は避けられませんが、その影響を最小化するためのさまざまな手法 (例: 先読み、予測、補間) があります。いくつかの手法については後ほど説明します。

その他のネットワーク用語

待ち時間に関するその他の用語をいくつか紹介します。

ピン:ネットワークの応答性を測定するために、基本的なメッセージを送受信するプロセスです。ラウンドトリップタイム (RTT) を簡略化したものと考えてください。

ジッター:これはネットワークの状態が不安定な場合に発生する RTT の変動です。待ち時間の軽減に影響を与え、パケットが順序どおりに届かなくなる可能性があります。

帯域幅:一定時間内にネットワーク経由で伝送できるデータ量です。大量のステートデータを伝送する必要があるゲームでは、帯域幅の拡大が重要になります。

これらの用語については、こちらのマルチプレイヤーネットワーキングの用語のページで詳しく説明しています。

ネットワーク同期

同期を保つために、クライアントとサーバーは継続的にメッセージを交換し、すべてのプレイヤー間で一貫したゲームステートを維持します。通常、クライアントはユーザーコマンドを高い頻度 (多くの場合は 60 Hz、または約 16 ミリ秒ごと) でサーバーに送信します。これらのコマンドには、アクションや入力 (例: マウスやゲームパッドの動き、ジャンプや射撃のボタン押下) が含まれます。

サーバーはクライアントコマンドを受信して処理したら、ゲームワールドに関する更新情報をクライアントに返します。ゲームがプレイヤー入力に反応する時間が速いほど、応答がより良く感じられます。

サーバーのティックレート、クライアントの更新レート、クライアントのフレームレートはそれぞれ目的が異なるため、必ずしも一致させる必要はありません。実際、完全な同期を実現することはまれです。サーバーとクライアントは、絶えず変動し続ける動的データの連続ストリームを交換しています。その目的は、ズレを減らして、すべてのクライアントが同時にプレイしているように見せることです。

ネットワーク同期の手法

ステート同期は、ネットワークオブジェクトのステートをサーバーからクライアントに定期的に伝送する手法です。ゲームの更新頻度は、そのゲームジャンル固有のニーズによって異なります (例 : 対戦型シューティングゲームと協力型ストラテジーゲームで求められる頻度は異なります)。

リモートプロシージャコール (RPC) は、サーバーまたは他のクライアント上の機能をリモートで呼び出します。RPC は、プレイヤー入力の送信、特定のアクションのリクエスト、一度のみ発生するゲームイベントのトリガーなど、クライアントとサーバー間の通信に使用します。

帯域幅の管理は、パフォーマンスに大きく影響する場合があります。同期は帯域幅を消費するため、以下のような戦略を実施してネットワークでのデータ伝送を減らします。

- **データカリング** : 不要な更新を省き、ゲームプレイに必要な情報のみに焦点を当てることで、ネットワークトラフィックを削減します。 例えば、重要な移動軸のみを同期したり、VFX とアニメーションを継続的に同期するのではなく、イベントを使用してそれらをローカルでトリガーしたりできます。ネットワークトラフィック削減はゲームパフォーマンスの向上につながります。
- **デルタ圧縮**: これはデルタエンコーディング (差分符号化) とも呼ばれ、サーバーは最後の更新以降の変更 (デルタ) のみを送信できます。その後、クライアントはこれらのデルタ情報のみをローカルのゲームステートに適用し、サーバーとの同期を維持します。
- **インタレスト管理** : 複数の基準に基づいてデータ同期に優先順位を付けます。空間の関連性では、プレイヤーからの距離と可視性に基づいてオブジェクトの優先順位が決定されます。経過時間 (または古さ) では、最近伝送されていないオブジェクトやデータに優先順位を付け、更新されるまで高い優先度が与えられます。インタラクションでは、プレイヤーが最近インタラクションを行ったオブジェクトや、これからインタラクションを行いそうなオブジェクトに優先度を与えます。

これらの手法は、ネットワークパフォーマンスを最適化し、より滑らかで効率的なゲームプレイ体験を実現するのに役立ちます。

ネットワークトポロジー

簡単に言うと、ネットワークトポロジーとは、マルチプレイヤー環境でのデバイスの接続方法や通信方法を定義するものです。各ネットワークモデルにはそれぞれ長所と短所があります。どれを採用するかは、ゲームの種類、ゲームステートに対する望ましい制御レベル、サーバーインフラストラクチャに使用できるリソースによって異なります。

トポロジーは、ゲームのアーキテクチャ、パフォーマンス、プレイヤー体験全体に影響を与える可能性があります。Netcode for GameObjects は、クライアントサーバー型と分散型権限の 2 つの主要なトポロジーに対応しています。それぞれが持つ意味を詳しく説明していきます。

クライアントサーバー型トポロジー

クライアントサーバー型トポロジーは、一般的なネットワークワークモデルの 1 つで、パフォーマンスを最適化してゲームを効果的に管理するために、クライアントデバイスと中央サーバーで役割を分担させます。

クライアントはプレイヤーのゲームインスタンスであり、ローカル入力、レンダリング、ゲームステートの部分的なシミュレーションをハンドルします。クライアントは、キャラクターの動きなどのローカル入力をサーバーに送信し、サーバーから更新情報を受け取ります。

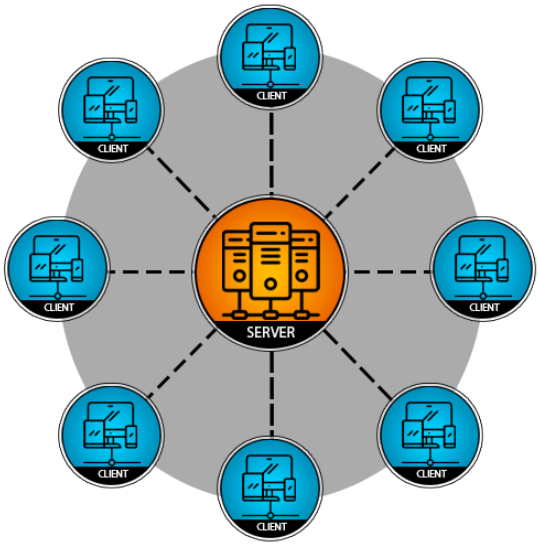
サーバーは、ゲームワールドを明確かつ正確に表現し、プレイヤー入力を処理してゲームルールを適用します。この中央サーバーは競合を解決し、アクションを検証して、すべてのプレイヤーに一貫した公平な体験を提供します。また、ゲームステートを一元的に制御することで、チートを防止するのにも役立ちます。

クライアントとサーバーは、インターネットまたはローカルネットワークワーク（LAN）を経由して相互に通信できます。オフライン LAN ゲームでは、同じ物理環境内にある複数のデバイスをローカルネットワーク経由で接続するため、インターネットにアクセスする必要はありません。この設定ではインターネットをバイパスし、デバイスが近接しているため、待ち時間を最小限に抑え、セキュリティに優れた信頼性の高い接続を実現します。そのため、LAN パーティや e スポーツ大会、インターネットが不安定な環境に適しています。

クライアントサーバー型トポロジーには、以下の 2 種類のサーバーがあります。

専用ゲームサーバー

専用ゲームサーバーは、データ処理のみを行い、プレイヤーとして参加しない独立したエンティティです。ネットワークを介したゲームでの主要なシミュレーションやプレイヤーのインタラクションをすべてハンドルしながら、最高のパフォーマンスを実現できます。



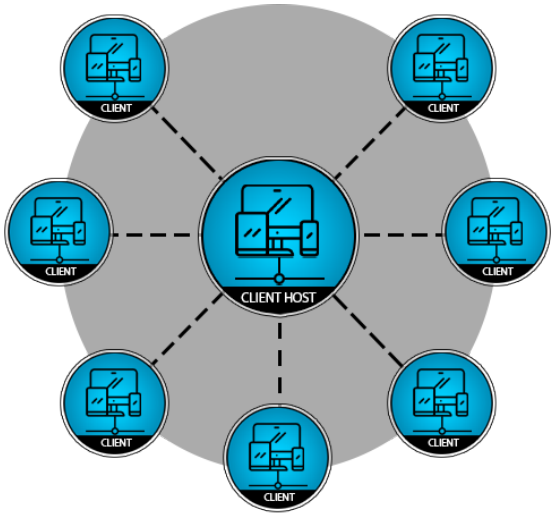
専用ゲームサーバーは、すべての主要なゲームシミュレーションをハンドルします。

専用サーバーは、チートを最小限に抑えることが最も重要なゲームにとって不可欠です。ただし、この設定ではすべてのプレイヤーステートの変更をサーバーで処理してから、他のクライアントにリレーする必要があるため、通信の待ち時間が発生する可能性があります。

専用サーバーは、一人称シューティングゲームなど、パフォーマンスが重視される対戦型ゲームに特に適しています。また、公平性を維持し、迷惑動作を減らすためにも不可欠です。

クライアントホスト型リッスンサーバー

クライアントホスト型リッスンサーバーは、サーバーとクライアントの両方として機能し、ホストがゲームをプレイできるようにします。これは、コストの削減に役立つほか、ネットワークワーク経由でパケットを送信する必要がないため、ホストの待ち時間の面でも有利になります。



この設定では、同じマシンでゲームサーバーの実行とホストプレイヤーのビジュアル出力の生成が行われるため、サーバーのパフォーマンスが低下することがよくあります。また、ホストされたクライアントは家庭用のインターネット接続を経由して接続を行うため、リモートデータセンターで専用サーバーを使用するよりも低速になることがあります。これは、家庭用のインターネットサービスプロバイダーは一般的に、アップロードのパフォーマンスよりもダウンロードのパフォーマンスを優先するためです。

クライアントホスト型リッスンサーバーは、サーバーとクライアントの両方として機能します。

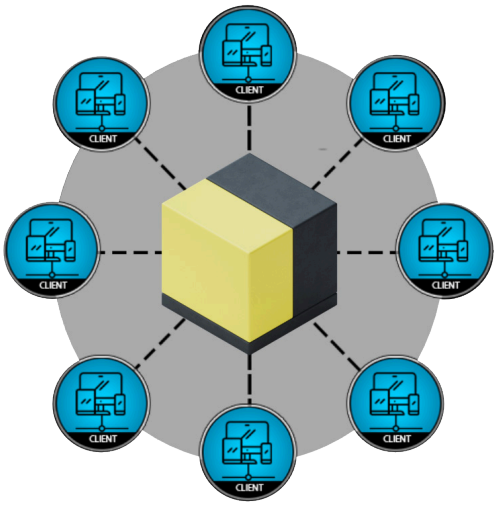
分散型権限

分散型権限ネットワークモデルでは、ゲームステートの制御と管理を、参加しているすべてのクライアント間で分散します。各クライアントは、ゲーム内のオブジェクトステートの一部を所有、追跡、管理する責任を持ち、そのオブジェクトを自律的にスポーンおよび管理することが可能です。

一元化された軽量のサービスは、オブジェクトステートの変化を監視し、ネットワークトラフィックのルーティングを管理しますが、ゲーム自体のシミュレーションは行いません。

このトポロジーには、コストの削減や入力待ち時間の削減など、いくつかの利点があります。なぜなら、各クライアントが独自のオブジェクトに対して権限を持つことで、すべてのゲームアクションを中央サーバーで処理する必要がなくなり、ネットワークのラウンドトリップが減るためです。例えば、移動や攻撃といったゲーム入力アクションは、サーバーからの許可を待つことなくローカルでハンドルできます。これにより、プレイヤーにより即時のフィードバックが得られ、ゲームの応答性が向上します。ただし、すべてのアクションを検証する単一の権威サーバーがないため、チートに対する脆弱性が高まる可能性もあります。

分散型権限は、精密なシミュレーションや高い競争性を必要とするゲームには不向きですが、インタラクションの重要度がそれほど高くないゲームには適しています。



分散型権限モデルは、制御とゲーム管理を分散させます。

Unity の新しい Netcode for GameObjects 向け Distributed Authority パッケージ（ベータ版）の詳細については、こちらの Unite 2024 セッション (YouTube 自動翻訳字幕推奨)を参照してください。

ローカルまたはカウチマルチプレイヤー

ローカルのマルチプレイヤーゲームでは、1 つのクライアントランタイムインスタンスを使用して、同じ画面上、同じ物理的な場所で 2 人以上のプレイヤーがプレイできます。この設定は、ネットワーキングを必要とせず、プレイヤー同士が直接相互作用できるため、ソーシャルゲームに最適です。パーティゲームや協力プレイモードでよく使用され、友人や家族と一緒に遊べる、シンプルでわかりやすい方法です。

ピアツーピア (P2P)

各デバイスがクライアントとサーバーの両方として機能し、プレイヤー同士を直接接続できます。これは分散型権限と似ていますが、軽量なサーバーすら持ちません。この方法により、中央サーバーが不要となり、コストと複雑さが軽減されます。ただし、ゲームステートやセキュリティを管理する中央の権限が存在しないため、公平性と一貫した待ち時間を確保するうえで課題が生じる可能性があります。

権威サーバーとは

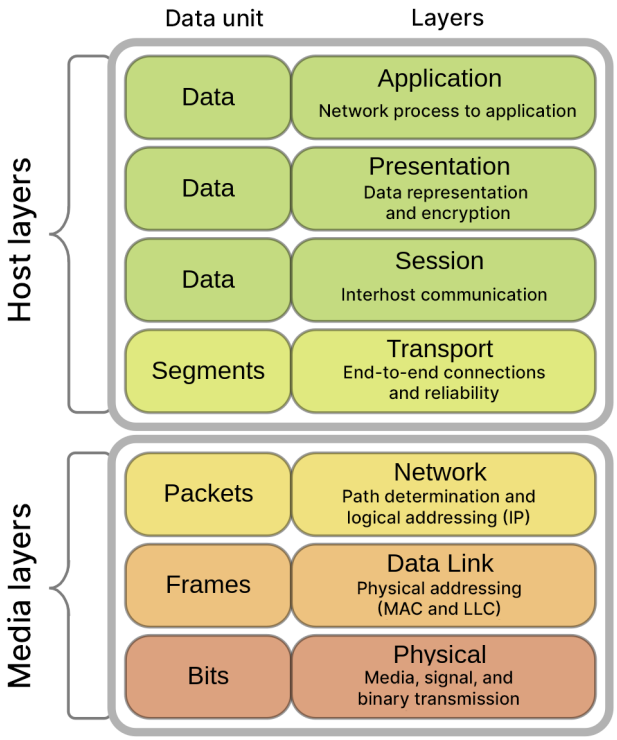
権威サーバーとは、ゲームステートやロジックの中央コントローラーであるサーバー設定のことです。その名の通り、ネットワークを介したゲームにおける最終的な権限を持つ存在です。

ゲームの進行や状態の権限をプレイヤーのマシンに分散させるのではなく、権威サーバーがゲームシミュレーション全体を実行し、ゲーム内で起きていることを決定します。クライアントは入力をサーバーに送信するだけで、サーバーがゲームを更新し、最新のゲームステートをクライアントに返します。

また、サーバーはゲームルールの適用や、競合の解決も行います。権威サーバーは、ネットワークを介したゲームのロジックを実装するうえで最もシンプルな方法の 1 つであり、チートによる悪用が最も起こりづらいサーバーの 1 つです。そのため、すべてのゲームプレイヤーに対して一貫性のある体験を提供できます。

ネットワークスタック

プロトコルスタック（ネットワークスタック）とは、一般的に、ネットワークを経由したデータ伝送を可能にする、さまざまな通信プロトコルを実装するソフトウェアを指します。この構成は、レイヤーケーキに例えられます。



ネットワークスタック（出展：Wikipedia）

各レイヤーは、直上と直下のレイヤーとのみ相互作用するため、モジュール性が確保され、ネットワークの管理がシンプルになります。

アプリケーションレイヤー：スタックの最上位レイヤーで、ほとんどの Unity 開発がここで行われます。Netcode for GameObjects や Netcode for Entities などのパッケージは、下位レベルのネットワーキングの複雑さを抽象化するため、開発者はマルチプレイヤー機能の実装に集中できます。

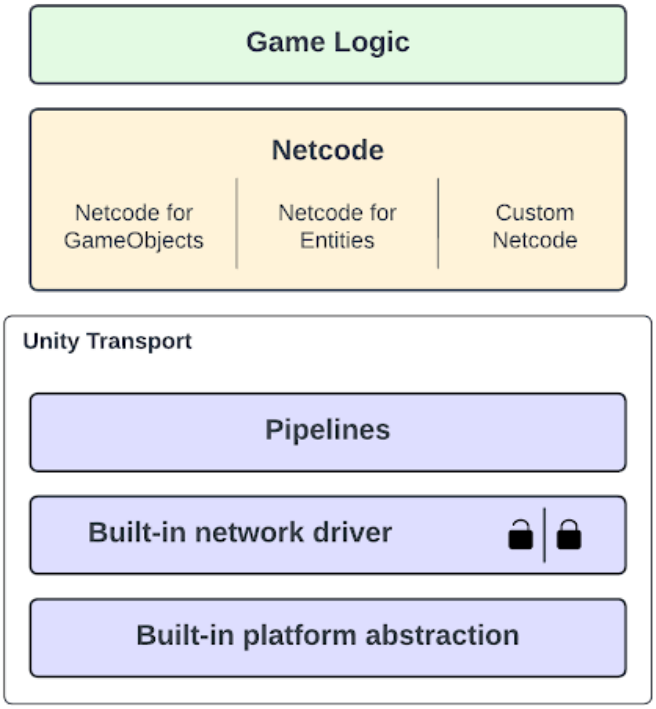
トランスポートレイヤー：トランスポートレイヤーは、信頼性の高いデータ転送、エラーの検出と修正、フロー制御、ネットワーク内におけるデバイス間のエンドツーエンドの通信確保を担います。データパケットのセグメント化とリアセンブリを容易にし、エラー回復とデータ整合性のメカニズムを提供します。

ネットワークレイヤー：ネットワークレイヤーは、異なるネットワーク上におけるネットワークに接続されたデバイス間でのデータパケットのルーティングを担います。IP（インターネットプロトコル）などのネットワークインフラストラクチャとプロトコルに依存して通信をハンドルします。

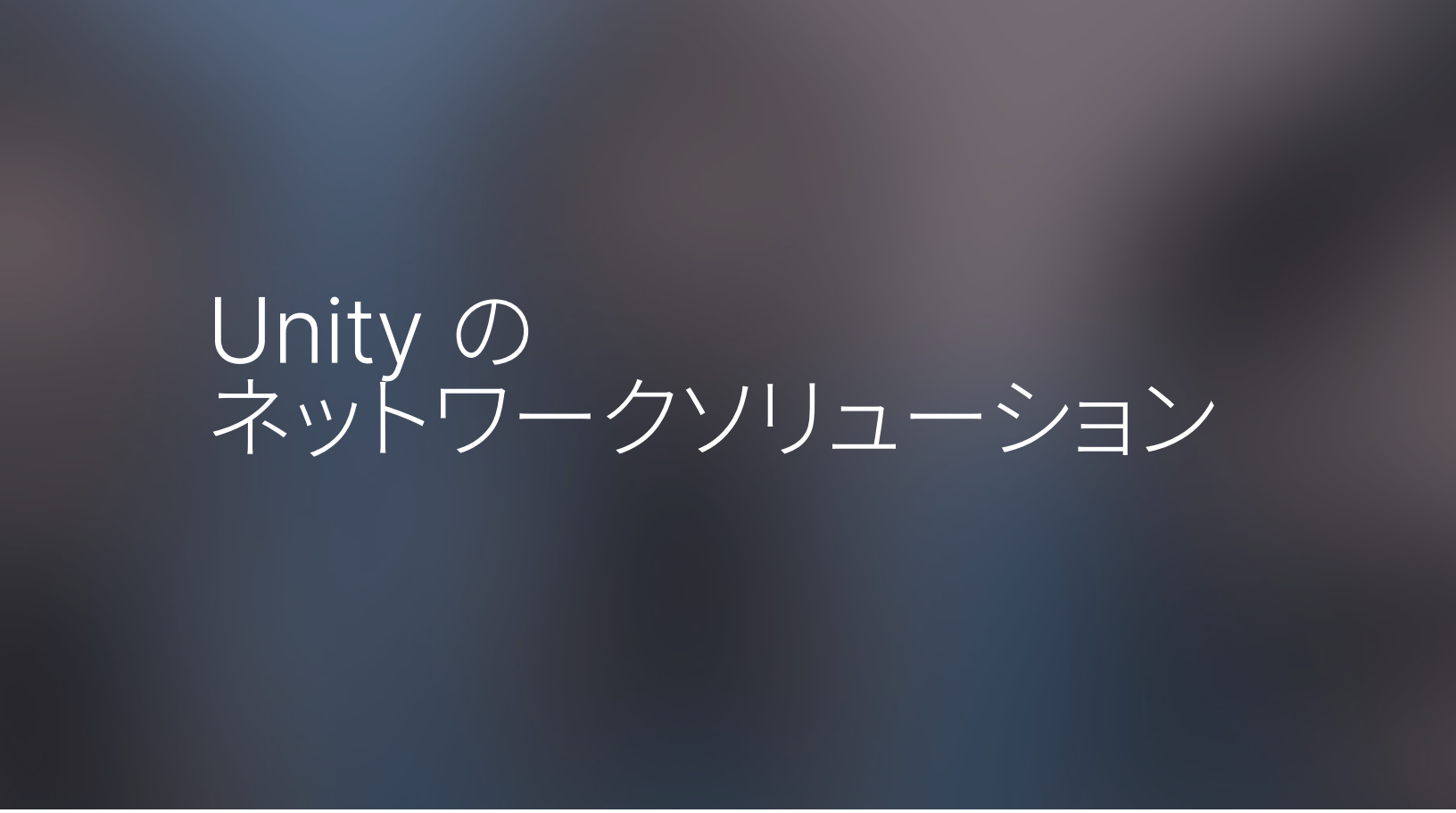
データリンクレイヤーと物理レイヤー : これらのレイヤーは、Ethernet や Wi-Fi などのネットワークメディアを介したデータパケットの物理伝送を直接ハンドルします。データリンクレイヤーと物理レイヤーは通常、オペレーティングシステムやネットワークワークハードウェアによってハンドルされます。

Unity 開発者は、まず上位レベルのアプリケーションレイヤーを使用して、ゲームオブジェクトの同期、ゲームステートの管理、プレイヤーインタラクションのハンドルといったマルチプレイヤー機能を実装します。

通常、アプリケーションに特定の要件がない限り、下位レイヤーについては気にする必要はありません。そのため、ネットワークスタックは以下のように簡略化されます。



NetCode 開発におけるレイヤー



Unity のネットワークソリューション

Unity は、さまざまなジャンルやスケールのマルチプレイヤーゲーム開発に対応する、包括的なソリューションを提供しています。ゲームの種類ごとに固有のネットワーキング要件が存在するため、適切なネットコードソリューションを選択することは非常に重要です。

ここでは、ゲームのジャンル、スケール、競争性のレベル、ネットワーキングレイヤーに対する望ましい制御など、いくつかの要素を考慮する必要があります。カジュアルゲームではシンプルさと費用対効果が優先されることが多く、対戦型ゲームでは公平性と応答性を確保するために精密かつ堅牢なネットワーク管理が必要です。

協力型カジュアルゲームでも対戦型アクションゲームでも、Unity はそれぞれのニーズを満たす強力なネットコードパッケージと補完的なサービスを提供しています。

Netcode for GameObjects

協力型のカジュアルなマルチプレイヤーゲームの場合、Netcode for GameObjects (NGO) パッケージの使用を推奨します。この高レベルのソリューションは、ネットワーキングのロジックを抽象化することでマルチプレイヤーゲームの開発をシンプルにし、全プレイヤーのゲームステートの管理を容易にします。

マルチプレイヤー開発を始めたばかりのユーザーにとって、NGO は出発点として最適です。

Netcode for GameObjects の主な機能は以下のとおりです。

— **NetworkObject:** これは、ネットワーク上で同期する必要があるゲーム内のオブジェクトを表します。オブジェクトのスポーン、デスポーン、所有権などをハンドルします。

- **NetworkBehaviour:** これは、ネットワーク機能を提供する特殊な MonoBehaviour です。ネットワークワークイベントをハンドルするコールバックがビルトインされており、サーバー側およびクライアント側のコードを記述できます。
- **リモートプロシージャコール (RPC):** これは、サーバー RPC とクライアント RPC を使用して、ゲームオブジェクトのリモートインスタンスにメッセージを送信したり、メソッドを呼び出したりします。
- **NetworkVariable:** これは、ネットワーク間でステートを同期できるクラスです。
- **NetworkManager:** これは、ゲームのネットワークステートを管理する中央コンポーネントで、接続、接続切断、シーン管理などのタスクをハンドルします。

これらのコンポーネントはアプリケーションレイヤーで機能し、マルチプレイヤー機能の実装に役立ちます。Netcode for GameObjects は、シンプルでありながら高機能に設計されており、堅牢でスケーラブルなマルチプレイヤーゲームの作成に必要なツールを提供します。

Netcode for Entities

Unity の Data-Oriented Technology Stack (DOTS) と Entity Component System (ECS) をベースにビルドされた Netcode for Entities は、サーバー権威型のゲームプレイ向けに設計されています。このソリューションには、クライアントサイド予測、補間、ラグ補償などの上級者向けの機能が用意されています。

この設定では、中央サーバーがすべてのゲームシミュレーションをハンドルし、ゲームの結果を制御することでチートを減らします。クライアントは入力をサーバーに送信し、サーバーがそれを処理して、更新されたゲームステートをクライアントに返します。これにより、不正な操作の可能性を最小限に抑えます。

Netcode for Entities には、待ち時間を軽減するクライアントサイド予測が搭載されており、より滑らかで、ラグがほとんど発生しないプレイ体験を実現します。Netcode for GameObjects と比較して、スケーラビリティと帯域幅の最適化に優れています。

経験豊富なマルチプレイヤー開発者であり、高度なパフォーマンスや決定論を必要とするプロジェクトに携わっている場合、ゲームのベースを DOTS と ECS にするのが適しているでしょう。

Unity におけるデータ指向設計の詳細については、DOTS の eBook および DOTS のリソースリストを参照してください。

Unity のサンプル『ECS Network Racing』は、Netcode for Entities を使用してビルドされています。

適切な NetCode ソリューションの選択は、プロジェクトの要件とチームの専門知識によって異なります。選び方の参考として、まずは以下の表を活用してください。

機能	Netcode for GameObjects	Netcode for Entities
対象オーディエンス	初級および中級開発者	上級開発者
建築設計	オブジェクト指向 (MonoBehaviour ベース)	データ指向 (Entity Component System - ECS および DOTS)
パフォーマンス	小規模のゲームに最適	高パフォーマンスおよび高スケーラビリティを要するゲームに最適
スケーラビリティ	制限あり (少人数プレイ向け)	高スケーラビリティ (大規模ゲーム向け)
ネットワーキング機能	NetworkVariable、RPC、NetworkTransform、限定的なクライアントサイド予測 (先読み)、補間、UnityTransport をサポート	フル機能のクライアントサイド予測、補間、ラグ補償、最適化された UnityTransport をサポート
Unity Services との統合	Lobby、Relay などをフルサポート	Lobby、Relay などをフルサポート
サンプルプロジェクト	Boss Room、Bite Size Samples	Megacity Metro、ECS Racing、Netcode サンプル

もう 1 つの優れたリソースが、Unity の Multiplayer Center です。Multiplayer Center は、マルチプレイヤーゲームを作成するための出発点となります。ゲームのニーズに応じて Unity のマルチプレイヤーパッケージを提案し、それらを活用する際に役立つサンプルやチュートリアルへのアクセスも提供されています。パッケージを入手して、プロジェクトで設定する手順については、Multiplayer Center のドキュメントを参照してください。

Unity Transport

Unity Transport パッケージは、ネットコードに依存しないライブラリで、パフォーマンスと信頼性に重点を置いた低レベルのネットワークレイヤーを提供します。これは、従来の UDP を上級者向けの機能で拡張した、安全で移植性の高い最新のトランスポートライブラリです。

- **信頼性 :**UDP 上に信頼性の高い通信を加え、TCP のオーバーヘッドを発生させずに重要なメッセージを配信
- **セキュリティ :** 暗号化と認証を組み込み、不正アクセスからデータを保護

— **パフォーマンス：**短い待ち時間と高いスループットを実現するために最適化

— **クロスプラットフォーム互換性：**異なるプラットフォームやデバイス間でシームレスに動作するよう設計

Netcode for GameObjects と Netcode for Entities はどちらも、デフォルトで Unity Transport を使用します。ネットワークのより詳細な制御を求める開発者は、Unity Transport をスタンドアロンのライブラリとして使用し、特定のゲームのニーズに合わせてカスタマイズした独自のネットコードをその上にビルドすることもできます。

Unity Transport では新たに WebGL がサポートされ、クロスプラットフォームおよびウェブマルチプレイヤー体験を強化できるようになりました。

サードパーティ製のネットワーク

Unity では Netcode for GameObjects を使用して開発を開始することを推奨していますが、これが唯一の選択肢ではないという点を理解しておくことが重要です。Unity コミュニティには、具体的なニーズに合わせて選べるさまざまなソリューションが用意されています。

Unity Asset Store から入手できるサードパーティ製のネットワーキングオプションをいくつか紹介します。

— **Photon Unity Networking (PUN):**Photon は、同時接続ユーザー数の多いゲームに適した、包括的でスケーラブルなソリューションを提供します。

— **Mirror:** シンプルで使いやすいと定評のある Mirror は、Unity の UNet をベースとした無料でオープンソースのネットワーキングライブラリです。

— **DarkRift Networking 2:** このソリューションは、高いパフォーマンスと優れた柔軟性を兼ね備え、ネットワーキングの詳細な制御を必要とする開発者向けにカスタマイズされています。

— **Forge Networking Remastered**: 詳細なカスタマイズと制御が可能なオープンソースのオプションです。

最初の Netcode プロジェクトを設定する

まだ Unity のネットワーキングソリューションを試したことがない場合、基本的な Netcode プロジェクトの設定に必要なネットワーキングパッケージをインポートしてから、必要なマルチプレイヤーコンポーネントを設定する必要があります。

この章では、Netcode for GameObjects を使用して、サンプルプロジェクトにネットワーキングを加える最初のステップについて説明します。Unity 6 では、Multiplayer Center を使用して新しいマルチプレイヤープロジェクトを設定したり、Multiplayer ウィジェットを使用して他の Unity Services をプロジェクトに統合したりできます。

始める前に

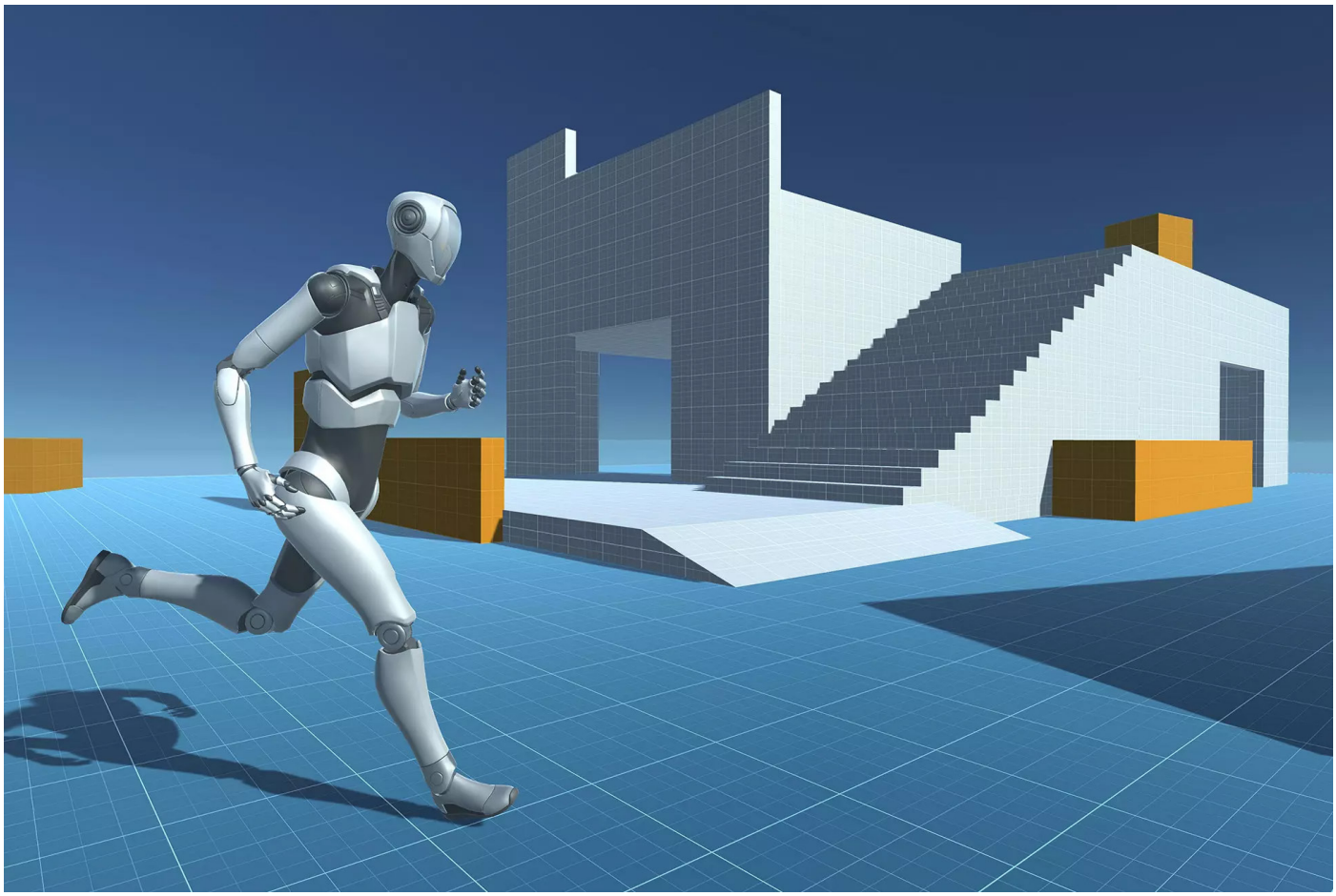
以下のものが揃っていることを確認します。

- 有効なライセンスを持つアクティブな Unity アカウント
- Unity Hub
- サポートされているバージョンの Unity エディター (ここで説明する一部の機能は、Unity 6 以降が必要です。Netcode for GameObjects の要件を参照してください)
- Unity Cloud ダッシュボードへの接続 (プロジェクトで必要な Unity Services に接続するには、Unity Hub を経由して Unity Cloud ダッシュボードに接続する必要があります)

サンプルプロジェクトの設定

前述のネットコードツールのデモを行うには、既存のプロジェクトでシングルプレイヤーの移動機能を使用すると効果的です。このガイドでは、Unity Asset Store にある Starter Assets – ThirdPerson パッケージを使用します。これは、ユニバーサルレンダーパイプライン (URP) を使用して、ヒューマノイドキャラクターによるシンプルな 3D ゲームプレイをシミュレートします。

この無料アセットを Unity Asset Store から入手し、Package Manager を使用してインポートします。



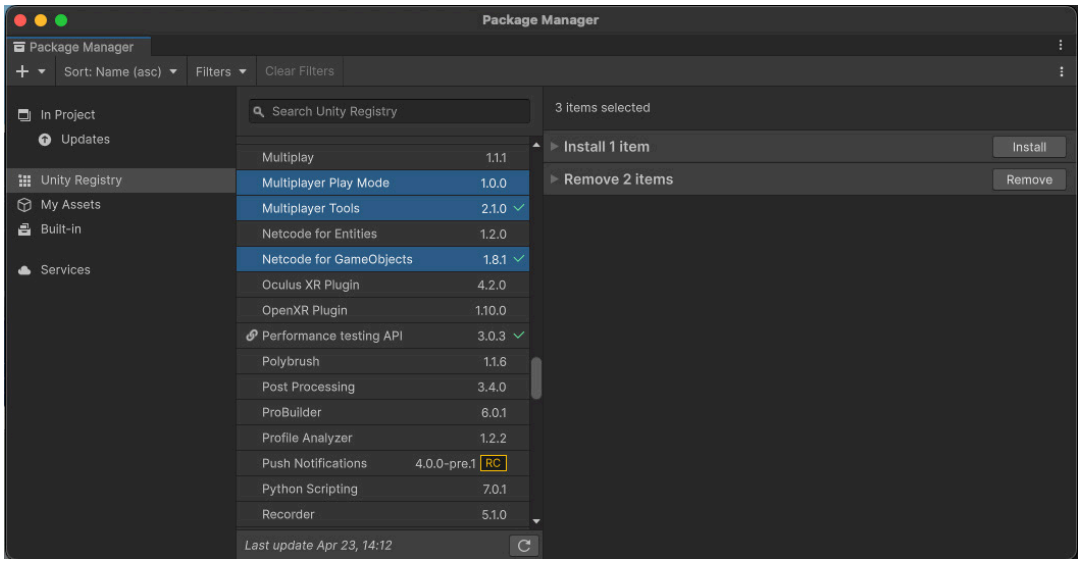
Asset Store の Starter Assets パッケージ

このデモプロジェクトには、小規模なテスト用のプレイグラウンドシーンと、設定可能な三人称視点のコントローラーが含まれています。目標は、このアプリケーションのコピーを複数実行し、異なるクライアントが同じ環境で相互作用できるようにすることです。

Netcode for GameObjects をインストールする

Package Manager (**Window > Package Manager**) で、Unity Registry を選択します。次に、以下のパッケージをインストールします。

- **Netcode for GameObjects**: これは、既存のゲームオブジェクト /MonoBehaviour ワークフローにマルチプレイヤー機能を加える基本的なネットワーキングライブラリです。マルチプレイヤーゲーム開発を効率化するもので、ネットワークを介したマルチプレイヤー開発を開始するのに最適です。
- **Multiplayer Tools ウィンドウ :** これは、Unity 6 で導入された 5 つの新たなツールスイートで、マルチプレイヤー開発ワークフローを改善します。
  - Multiplayer Tools ウィンドウを使用すると、すべてのマルチプレイヤーツールとそのドキュメントに 1 か所から簡単にアクセスできます。
  - Network Simulator は、パケット遅延、損失、接続切断など、実際のネットワーク状態を複製し、本番稼働の前に潜在的な問題を特定します。
  - Runtime Network Stats Monitor (RNSM) はリアルタイムのネットワーク統計を表示し、設定可能な画面でネットワークパフォーマンスを監視できます。
  - Network Scene Visualization は、ネットワークアクティビティとオブジェクト所有権をシーンビューに視覚的に表示し、デバッグを強化します。
  - Hierarchy Network Debug ビューでは、Hierarchy ウィンドウの右側にオーバーレイが表示され、ネットワークに接続されているオブジェクト (小さなネットワークキューブロゴ付き) を識別できます。
- Multiplayer Play Mode: この Unity 6 パッケージ を使用すると、Unity エディターを離れることなくマルチプレイヤー機能をテストできます。最大 4 人のプレイヤー (メインエディタープレイヤーと 3 人のバーチャルプレイヤー) をシミュレートできるため、プレイテストを高速化できます。



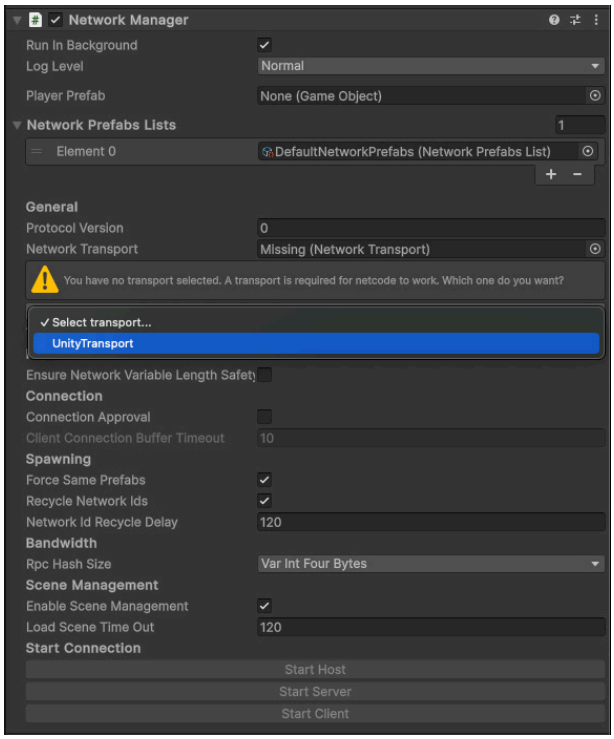
Netcode for GameObjects とそのサポートパッケージをインストールします。

NetworkManager を加える

すべてのプロジェクトには、ネットワークを介したマルチプレイヤーに対応するために **NetworkManager** コンポーネントが必要です。この必須コンポーネントはプロジェクトのネットワークステートを管理し、接続とネットワーク設定をハンドルします。

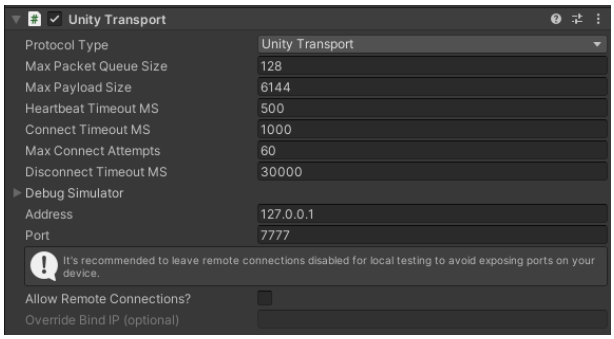
NetworkManager をシーンに加えるには、Hierarchy に新しいゲームオブジェクトを作成し、**NetworkManager** コンポーネント (**Netcode > NetworkManager**) を加えます。

NetworkManager コンポーネントで、**Network Transport** レイヤーを設定し、Unity Transport を選択します。



NetworkManager でトランスポートレイヤーを選択します。

これにより、ゲームオブジェクトに **UnityTransport** コンポーネントがアタッチされます。トランスポートレイヤーは、接続管理、データ伝送、パケット暗号化などの低レベルのネットワークタスクを担います。



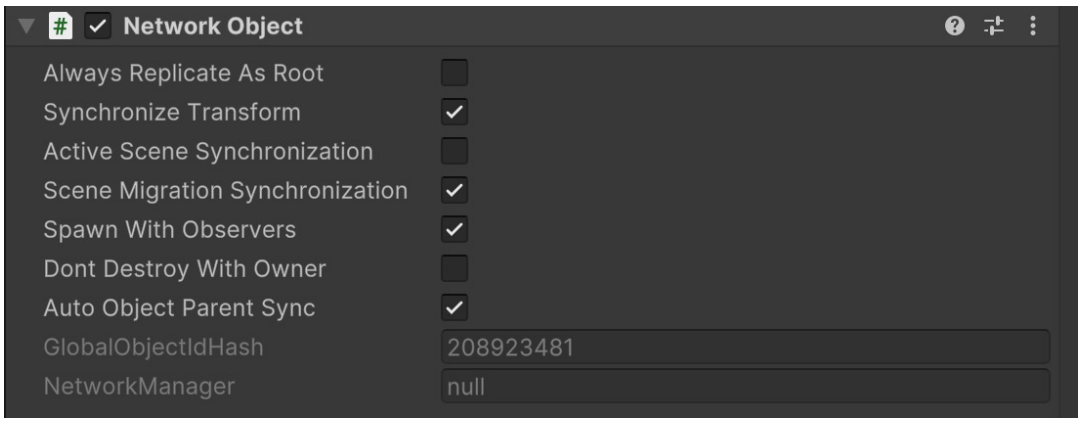
UnityTransport コンポーネント

この時点では、設定を変更する必要はありませんが、このコンポーネントは、ネットワークワーク状態 (例 : 待ち時間、パケットロス、ジッター) をシミュレートするために使用でき、エディターでのテストやデバッグに役立ちます。

シーンを保存して、**File > Build Settings** の順に移動して、現在のシーンが **Scenes in Build** リストに加わっていることを確認します。これにより、新しい NetworkManager がゲームビルドに含まれるようになります。

NetworkObject

NetworkObject は、マルチプレイヤーゲーム内のさまざまなクライアント間にわたってネットワーク接続または同期する必要がある、すべてのゲームオブジェクトに対して必須のコンポーネントです。ゲームオブジェクトに NetworkObject コンポーネントを加えると、そのコンポーネントは " ネットワーク対応可能 " になり、そのステートや動作をネットワークで共有および更新できるようになります。



NetworkObject コンポーネントとその一意の ID

各 NetworkObject には、いくつかの識別子があります。

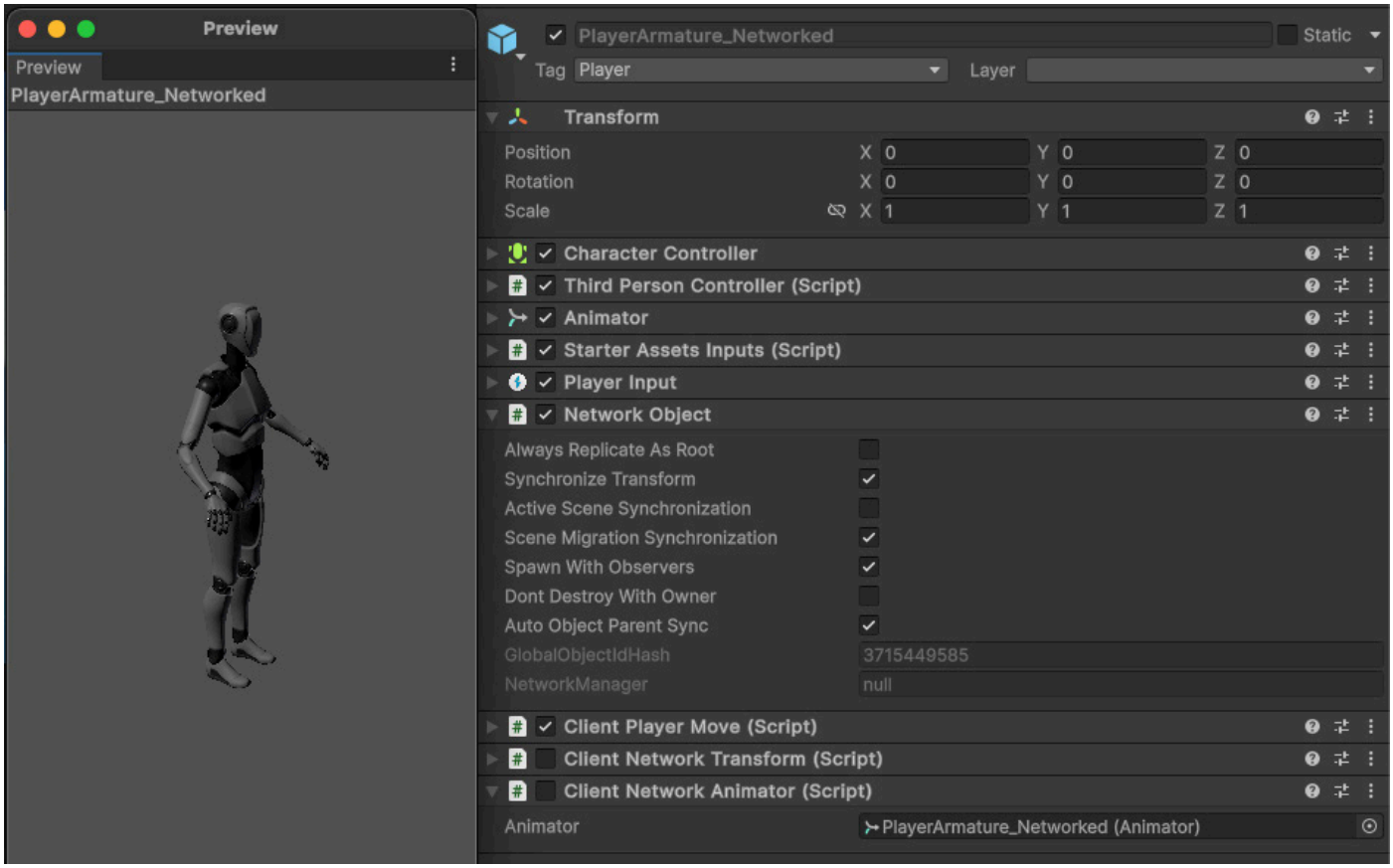
- **GlobalObjectIdHash** は、プロジェクト内のプレハブアセットを識別します。
- **NetworkObjectId** は、同じプレハブアセットのインスタンスを区別するための一意の識別子です。
- **OwnerClientId** は、オブジェクトを " 所有 " するクライアントを表します (以下の「権限」を参照してください)。

これらの識別子は、NetworkManager がオブジェクトを追跡し、接続されているすべてのクライアントにわたって、そのステートの一貫性を確保するのに役立ちます。NetworkObject は、ゲームプレイ中に動的に作成 (スポーン) または破棄できます。

NetworkObject をスポーンすると、接続されているすべてのクライアントに表示されます。各 NetworkObject には所有者が存在し、通常はその動作やステートを制御するクライアントが所有者です。

Player NetworkObject

各プレイヤーは、任意でプレイヤー用 NetworkObject と呼ばれる独自のプレハブを持つことができます。これは特殊なタイプの NetworkObject で、多くの場合、キャラクターコントローラーやゲーム内のプレイヤーのビジュアル表現が含まれます。



サンプルプロジェクトのプレイヤー用 NetworkObject

プレイヤー用 NetworkObject には多くの場合、プレイヤーの名前、スコア、インベントリ、その他の関連情報など、プレイヤー固有のデータが保存および同期されます。こうしたデータはネットワーク間で同期されるため、接続されているすべてのプレイヤーに一貫したゲームステートが表示されるようになります。

クライアントが接続すると、NetworkManager は、対応するプレイヤーが " 所有 " するプレイヤー用 NetworkObject を作成します。つまり、プレイヤーは自分の PlayerObject に対する権限を持ち、その動作やステートを制御できます。

プレイヤー用 NetworkObject を設定するには、まずプロジェクトで標準のプレハブゲームオブジェクトを作成します。このプレハブは PlayerObject のテンプレートとして機能し、プレイヤーの動作や外見を定義するために必要なコンポーネントとスクリプトが含まれています。

次に、適切なネットコードコンポーネントを加えます。その代表例は以下のとおりです。

- **NetworkObject:** ネットワーク接続を行うすべてのオブジェクトには、NetworkObject コンポーネントが必要です。これには、スポーン、デスポーン、所有権に関連するプロパティとイベントが含まれます。
- **NetworkBehaviour:**MonoBehaviour の基本クラスにネットワーク動作を加えるスクリプトです。NetworkBehaviour には、ネットワーク変数、リモートプロシージャコール (RPC)、ネットワークコールバックが含まれます。
- **NetworkAnimator:** このコンポーネントは、クライアント間でアニメーションステートとパラメーターを同期します。
- **NetworkTransform:** このコンポーネントにより、プレイヤーの位置、回転、スケールは、サーバーから接続されているすべてのクライアントにリアルタイムで複製されます。

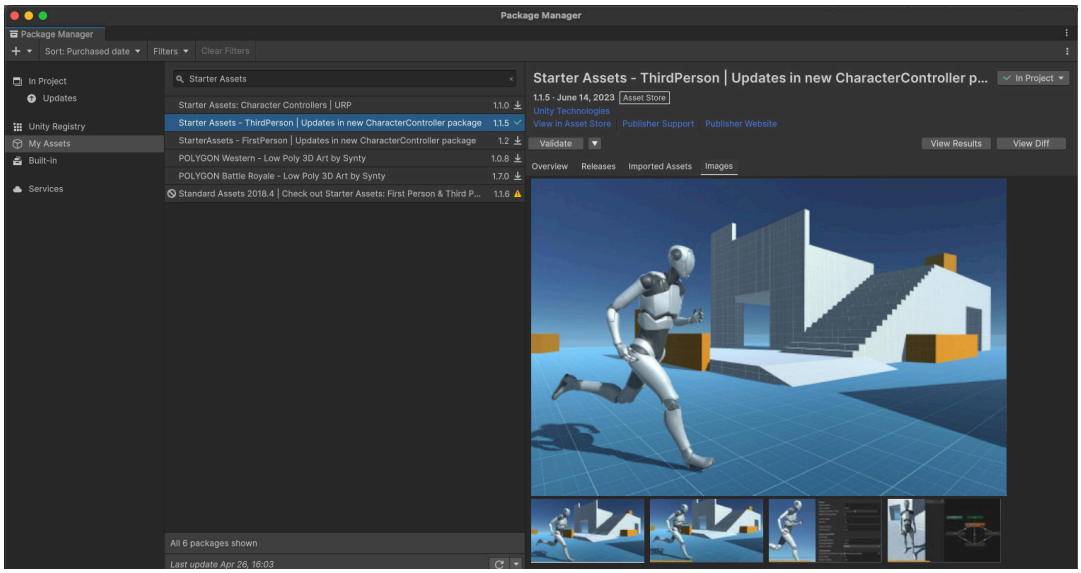
多くの場合、プレイヤー用 NetworkObject はプレイヤー入力のハンドルも担います。プレイヤーがゲームワールドで移動や相互作用などのアクションを実行すると、その入力が処理され、必要に応じて他の接続されたプレイヤーに反映されます。

プレイヤーのロジックは、ゲームメカニクスを直接制御する MonoBehaviour と、ネットワークステートを管理する NetworkBehaviour で構成されています。キャラクターコントローラーやアニメーターなど、ネットワーク接続に非対応のコンポーネントは通常、各プレイヤーのローカルインスタンスで機能します。

これらのコンポーネントをローカルで使用すると、パフォーマンスが最適化されるだけでなく、ネットワークトラフィックも削減されます。これは、クライアント間の帯域幅が制限されている場合に重要になります。

プレイヤー用 NetworkObject を作成する

サンプルプロジェクトから **Playground** シーンをロードします。

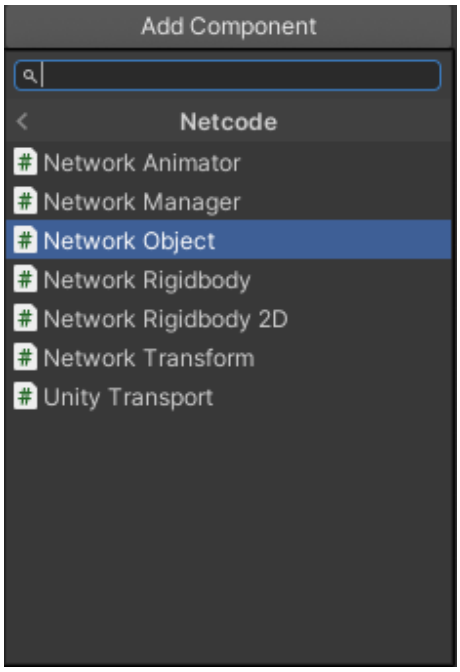


Starter Assets バンドルには Playground シーンが含まれています。

Hierarchy には、ゲームキャラクターの動作を制御する PlayerArmature があります。これをプレイヤー用 NetworkObject に変換するには、Hierarchy から PlayerArmature をドラッグして、新しいオリジナルプレハブを作成するか、プロジェクト内の既存のプレハブのコピーを編集します。

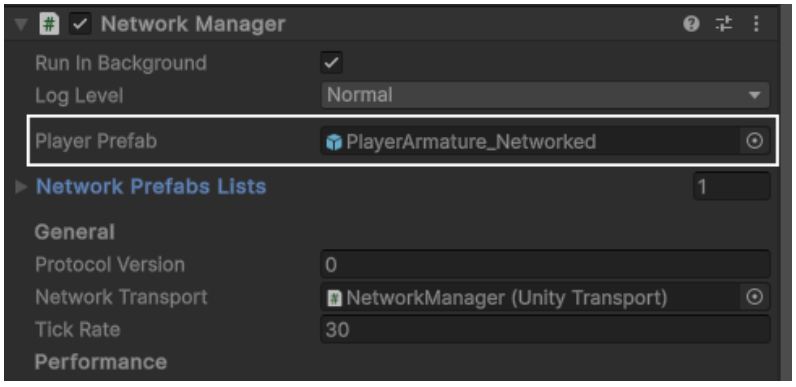
Hierarchy ウィンドウで PlayerArmature ゲームオブジェクトを見つけたら削除します。これにより、シーンから PlayerArmature とその子オブジェクトが除外され、ゲーム環境だけが残ります。

次に、Inspector でプレハブを編集します。**NetworkObject** コンポーネントを加えます。このコンポーネントは、オブジェクトがネットワーク経由で認識および管理されるために必要です。



プレハブに NetworkObject を加えます。

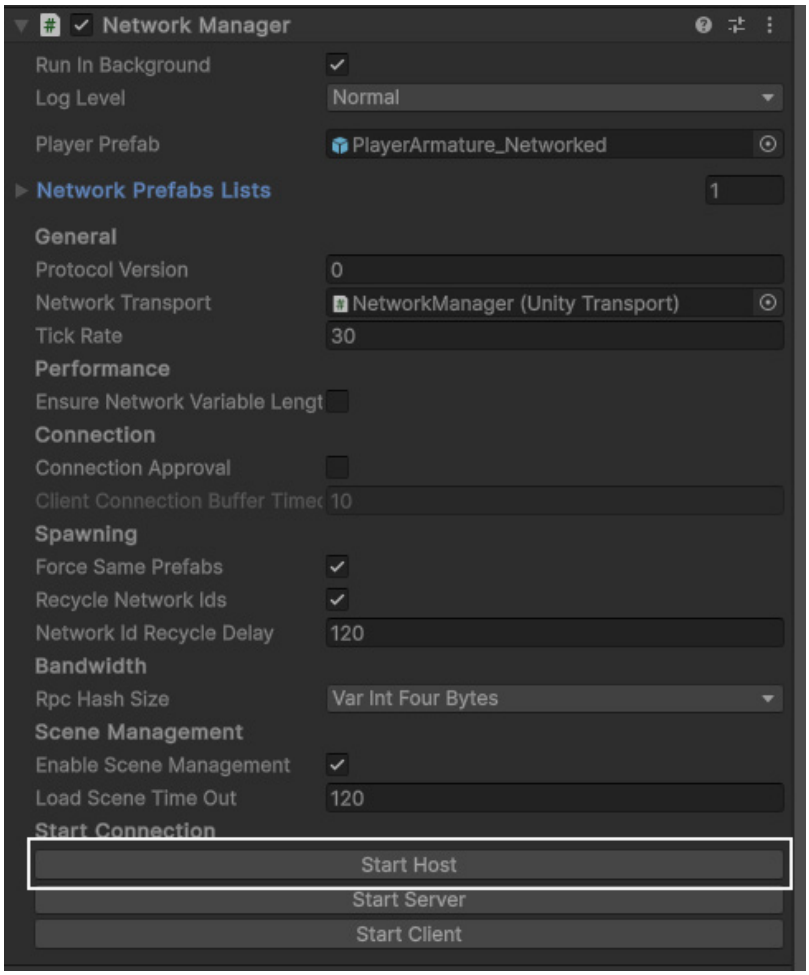
NetworkManager の **Player Prefab** フィールドにプレイヤー用 NetworkObject を登録します。



NetworkManager にプレイヤー用 NetworkObject を登録します。

再生モードには、ゲーム環境のみが表示されます。プレイヤー用 NetworkObject は、クライアントが接続された時にのみ表示されます。

NetworkManager を選択します。これは、Hierarchy で **DontDestroyOnLoad** 配下に表示されています。



NetworkManager でホストを開始します。

Start Host を選択します。これにより、Player NetworkedObject がスポーンされます。

PlayerArmature_Network というオブジェクトが Hierarchy に表示され、ゲームは再度プレイ可能になります (ただし、カメラのターゲットは無効です)。WASD コントロールを使用して、プレイヤーの動きをテストします。

再生モードを終了すると、プレイヤーキャラクターは消えます。NetworkManager は、ランタイムにこの特定のプレイヤーキャラクターをスポーンして管理するようになります。

複数のクライアントが接続されるまで、ゲームのネットワーク機能は確認できないことに注意してください。マルチプレイヤーシーンでの動作を理解するには、複数のクライアントでテストする必要があります。

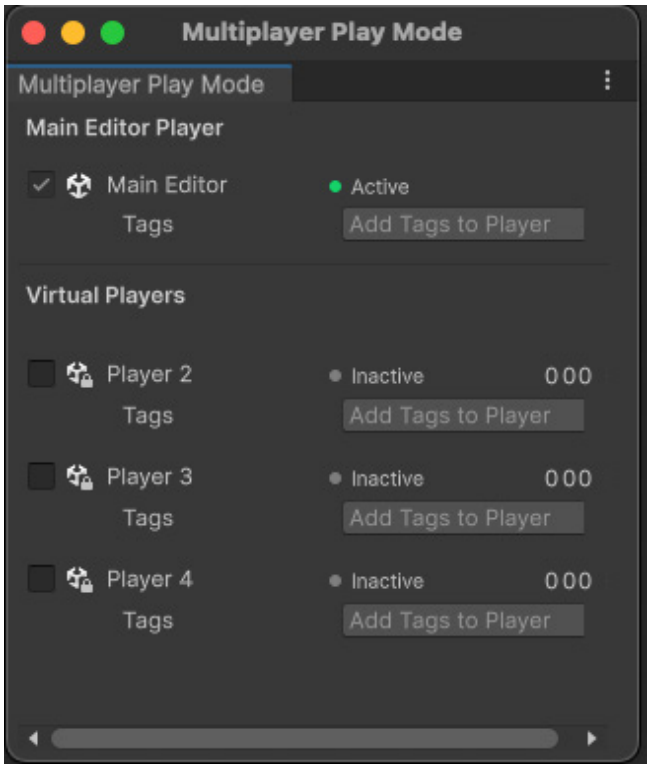
Multiplayer Play Mode

マルチプレイヤーをテストするには、アプリケーションを別々のランタイムプロセスで同時に実行する必要があります。以前までは、別途ゲームビルドを作成し、Unity エディターとは別に実行する必要がありました。

このオプションも引き続き利用できますが、Unity 6 には **Multiplayer Play Mode** (**MPPM**) が導入されています。MPPM を使用すると、開発者は Unity エディターの複数のインスタンスを同時に開き、マルチプレイヤー環境を複製できます。これにより、マルチプレイヤーのテストプロセスが効率化します。

Package Manager から Multiplayer Play Mode をインストールします。これにより、新しい機能をテストするたびにアプリケーションをビルドする必要がなくなります。

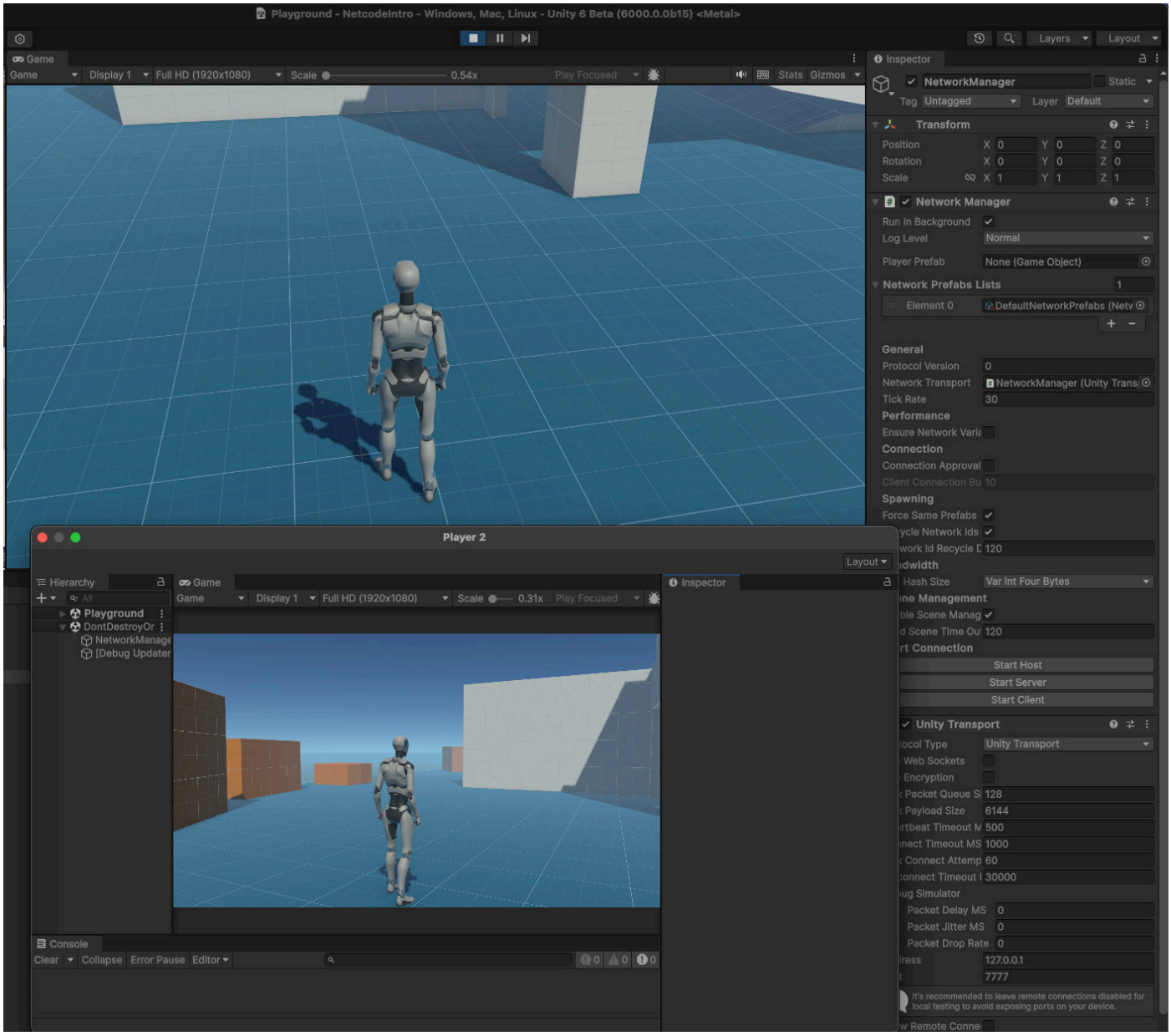
Multiplayer Play Mode (**Window > Multiplayer Play Mode**) を開きます。



Multiplayer Play Mode ウィンドウ

次に、上のスクリーンショットのリストから少なくとも 1 つの Virtual Player を有効にします。これにより、ホストとクライアントの最低構成 (各 1 つずつ) でテストできます。ホストは、クライアントであると同時にサーバーでも動作しています。

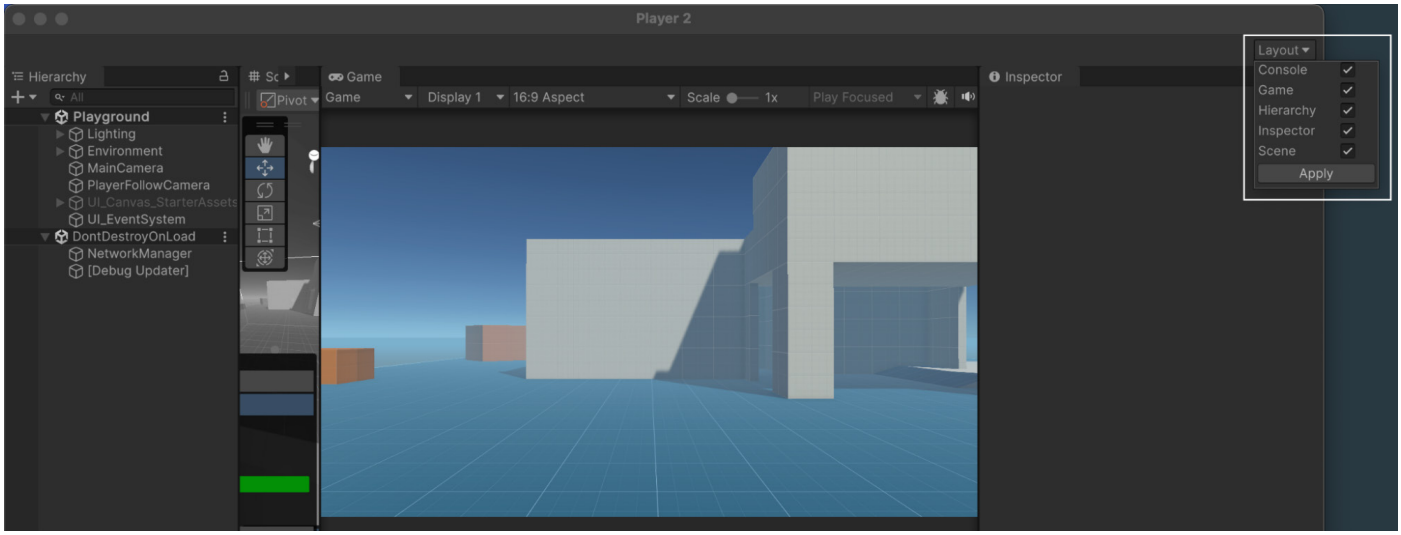
再生モードを開始すると、アプリケーションの 2 つ目のセッションがクローンウィンドウで実行を開始します。



Multiplayer Play Mode は、Virtual Player のクローンを作成します。

Hierarchy で **NetworkManager** を選択します。Inspector の **Start Connection** で、**Start Host** を選択します。PlayerArmature_Networked オブジェクトがゲームビューに表示されます。

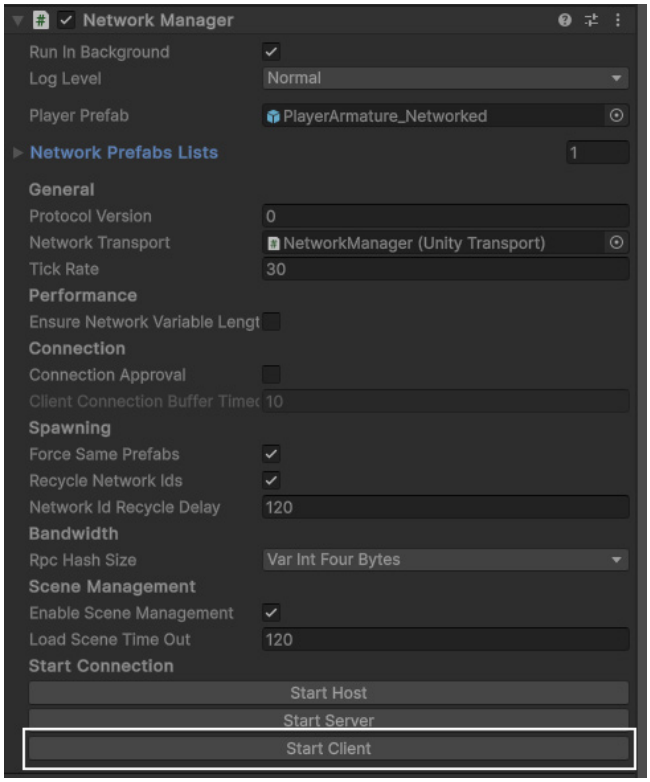
2 つ目のウィンドウにある Layout ボタンを使用し、Inspector や Hierarchy を有効にします。つまり、エディターをもう 1 つ起動したような状態にします。有効にするユーザーインターフェースコンポーネントを選択し、**Apply** をクリックします。



クローンウィンドウで Layout メニューを有効にします。

クローン側のエディターウィンドウで、Hierarchy の **DontDestroyOnLoad** 配下にある **NetworkManager** を探します。

Inspector ペインの **Start Connection** で、**Start Client を選択します。**



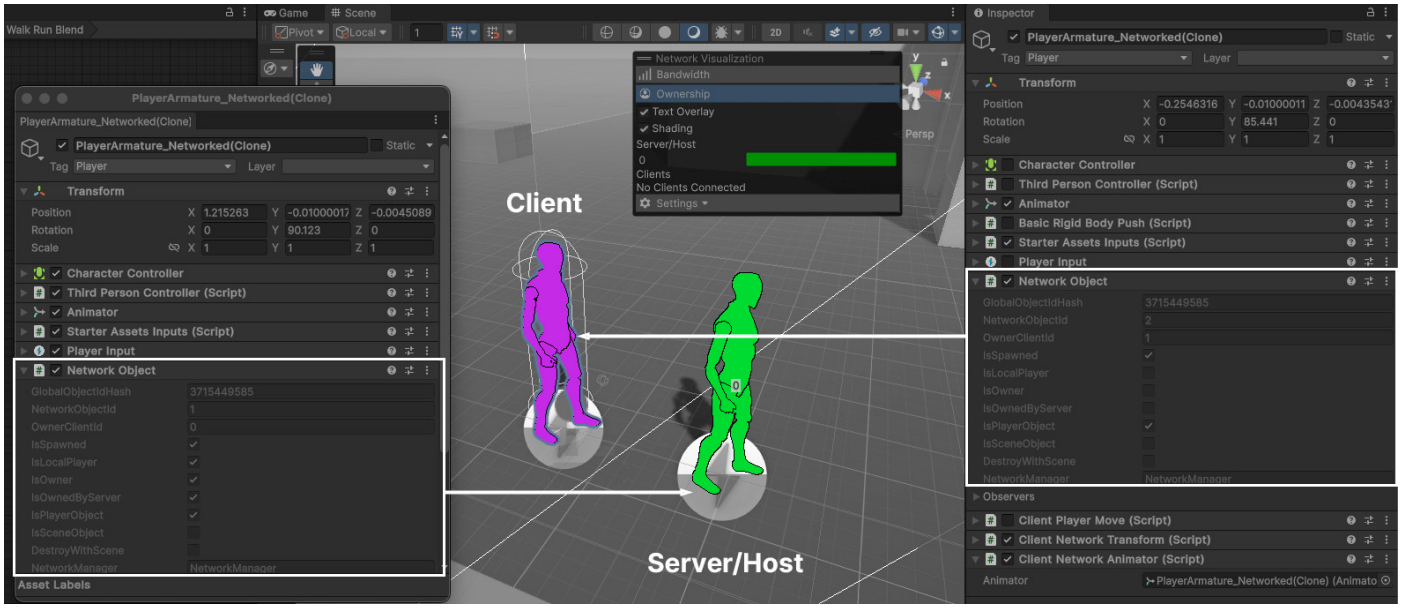
NetworkManager には、クライアントを接続するためのボタンがあります。

これで、Hierarchy には、PlayerArmature_Networked のインスタンスが 2 つ表示されます。シーンビューでは、それらが重なり合って表示されます。キーボードまたはゲームパッドを使用して、1 つのプレイヤーインスタンスをもう 1 つのプレイヤーインスタンスから離します。

Hierarchy で 1 つの PlayerArmature_Networke インスタンスを選択し、その NetworkObject コンポーネントを調査します。

ランタイム時に、各インスタンスが **GlobalObjectIdHash**（プロジェクトのアセット ID）と、**NetworkObjectId**（一意のインスタンスインデックス）によって識別されていることを確認します。これらの下にある **OwnerClientId** は、ホストとクライアントのどちらがインスタンスを制御しているかを示します。

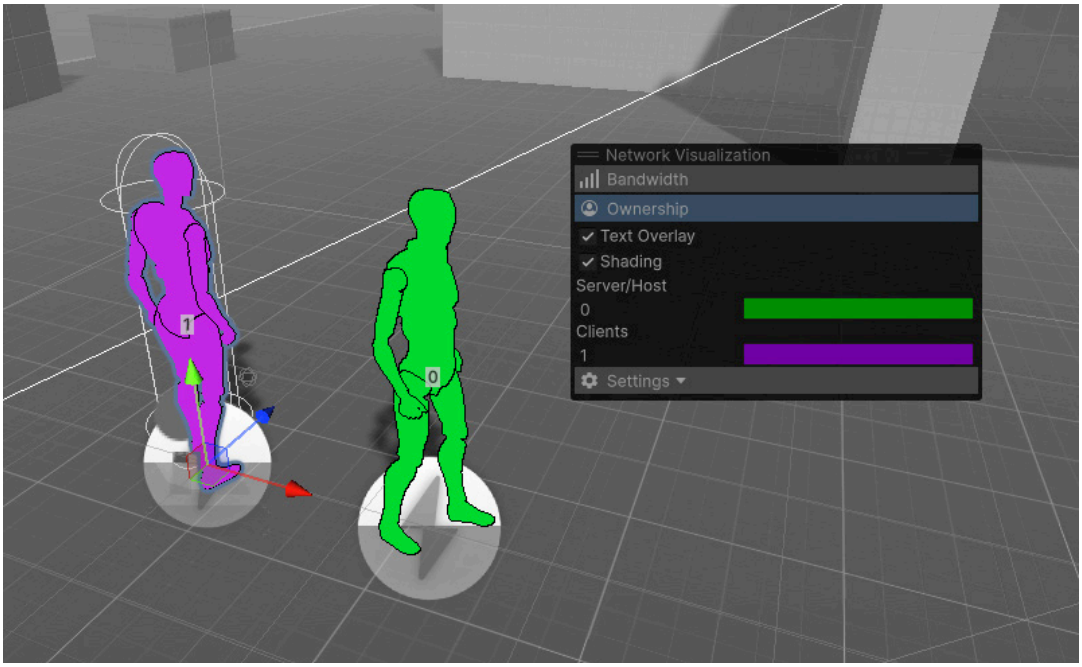
2 つのインスタンスを切り替えて、**IsSpawned**、**IsLocalPlayer**、**IsOwner**、**IsOwnerByServer** などのフラグの違いを比較します。



2 つのインスタンスの NetworkObject 設定を比較します。

2 つのインスタンスの違いをより明確にするには、Network Visualization パネルを使用します。この便利な診断ツールは、Multiplayer Tools パッケージをインストールするとシーンビューに表示されます。

2 つのインスタンスは、**Bandwidth**（どれだけのデータが伝送されているか）または **Ownership**（どのクライアントがプレイヤー用 NetworkObject に対する権限を持っているか）によって色分けされます。



Network Visualization は、ネットワークオブジェクトのデバッグに役立ちます。

NetworkManager はクライアントごとに別個のインスタンスを作成しますが、それぞれが同じキャラクターを独立して制御します。シーンビューでは、WASD コントロールによって制御されるキャラクターの動きは、クライアントとホスト間で同期されていません。

NetworkManager はプレイヤーの初回接続時に位置を座標 (0, 0, 0) で同期しますが、それ以降の動きは同期しません。現在、キャラクターの動作はいくつかのローカルコンポーネントによって制御されます。

- **CharacterController** を使用すると、複雑な物理演算を必要とせずに、ゲーム環境と相互作用しながらプレイヤーを移動させることができます。
- **Animator** は、ステートマシンに基づいてアニメーションを可能にします。走る、ジャンプする、待つといったステート間の遷移やブレンドを制御します。
- `PlayerInput` は、プレイヤーごとの入力管理、デバイスのペアリング、イベント通知をハンドルし、Unity Input System を扱いやすくまとめます。
- `StarterAssetsInputs` は、その入力をキャラクターの動き、視点操作、ジャンプ、ダッシュの入力用の値に変換します。

これらはシングルプレイヤー向けのコンポーネントです。マルチプレイヤーアプリケーションで機能させるには、ネットワーク対応のスクリプトをいくつか加える必要があります。

独自の UI 開始ボタンを作成する

ランタイム時にネットワークセッションをより簡単に開始できるように、NetworkManager の Inspector のボタン機能を複製するオンスクリーンボタンを作成できます。これには、Unity UI (UGUI) または UI Toolkit のどちらかを使用します。

任意の UI で、Client、Host、Server というラベルの付いた 3 つのボタンを作成します。次に、それぞれのボタンが NetworkManager シングルトンの以下のコールバックを呼び出すようにします。

- `NetworkManager.Singleton.StartClient`
- `NetworkManager.Singleton.StartHost`
- `NetworkManager.Singleton.StartServer`

これらのコールバックを使用すると、Inspector ウィンドウのボタンを使用せずにネットワークセッションを開始できます。

NetworkBehaviour を加える

PlayerArmature_Networked の MonoBehaviour を管理するために、NetworkBehaviour を使用できます。

`NetworkBehaviour` は、ネットワークを利用したロジック用に設計された特殊な種類の MonoBehaviour です。異なるゲームクライアント間でアクションやステートを同期するために必要なフレームワークを提供します。

`NetworkBehaviour` は、MonoBehaviour と同じライフサイクルイベントを共有しますが、ネットワーク固有の機能も複数組み込まれています。

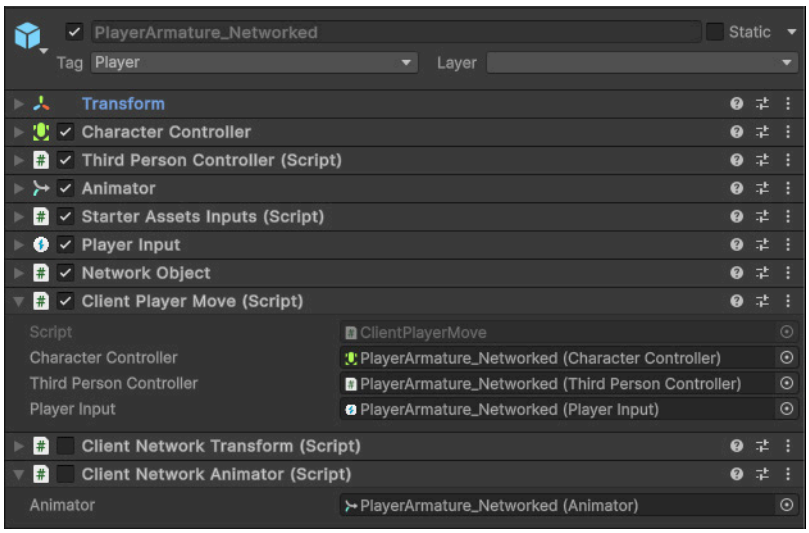
RPC メソッド : NetworkBehaviour では、リモートプロシージャコール (RPC) を使用して、ネットワークを経由した通信をハンドルできます。これらのメソッドには、[Rpc] 属性が付けられます。サーバーに送信するには [RPC(`SendTo.Server`)] 、クライアントに送信するには [RPC(`SendTo.Client`)] を呼び出します。

- `NetworkVariable`**:** これは、ネットワーク上でステートを同期管理するために設計された特殊な変数です。`NetworkVariable` に対するサーバー上での変更は、すべてのクライアントに自動的に反映されます。
- `OnNetworkSpawn` と `OnNetworkDespawn`: これらのライフサイクルメソッドは、NetworkBehaviour がインスタンス化または破棄されるとトリガーされます。`OnNetworkSpawn` は初期化に使用されます (OnEnable や Start に似ていますが、ネットワーク動作に対応したものと考えてください)。`OnNetworkDespawn` は、オブジェクトがネットワークから削除される前のクリーンアップをハンドルします (OnDestroy や OnDisable と似ています)。
- **Ownership**:`NetworkBehaviour` を使用すると、特定のクライアント (またはサーバー) が特定のオブジェクトに対して所有権を持つことができます。クライアントまたはサーバーのいずれかが NetworkObject を ” 所有” できるという権限の概念により、特定のオブジェクトの制御またはそれらとの相互作用が可能なのは、指定されたプレイヤーのみに制限されます。

ClientPlayerMove という NetworkBehaviour を実装して、プレイヤーの動きを管理できます。これにより、ホストおよびクライアントからの入力が、それぞれのプレイヤーオブジェクトに対してのみ機能するようになります。設定例は以下のとおりです。

```
using Unity.Netcode;
using StarterAssets;
using UnityEngine;
using UnityEngine.InputSystem;
namespace NetcodeDemo
{
   public class ClientPlayerMove: NetworkBehaviour
   {
       [SerializeField]
       CharacterController m_CharacterController;
       [SerializeField]
       ThirdPersonController m_ThirdPersonController;
       [SerializeField]
       PlayerInput m_PlayerInput;

       [SerializeField]
       Transform m_CameraFollow;
       private void Awake()

       {
           m_PlayerInput.enabled = false;
           m_ThirdPersonController.enabled = false;
           m_CharacterController.enabled = false;
       }

       public override void OnNetworkSpawn()
       {
           base.OnNetworkSpawn();
           enabled = IsClient; // クライアントの場合は有効にする
           if (!IsOwner)
           {
               // 所有者でない場合は無効にする
               enabled = false;
               m_PlayerInput.enabled = false;
               m_CharacterController.enabled = false;
               m_ThirdPersonController.enabled = false;
               return;
           }

           // 所有者の場合は有効にする
           m_PlayerInput.enabled = true;
           m_CharacterController.enabled = true;
           m_ThirdPersonController.enabled = true;

       }
   }
}
```

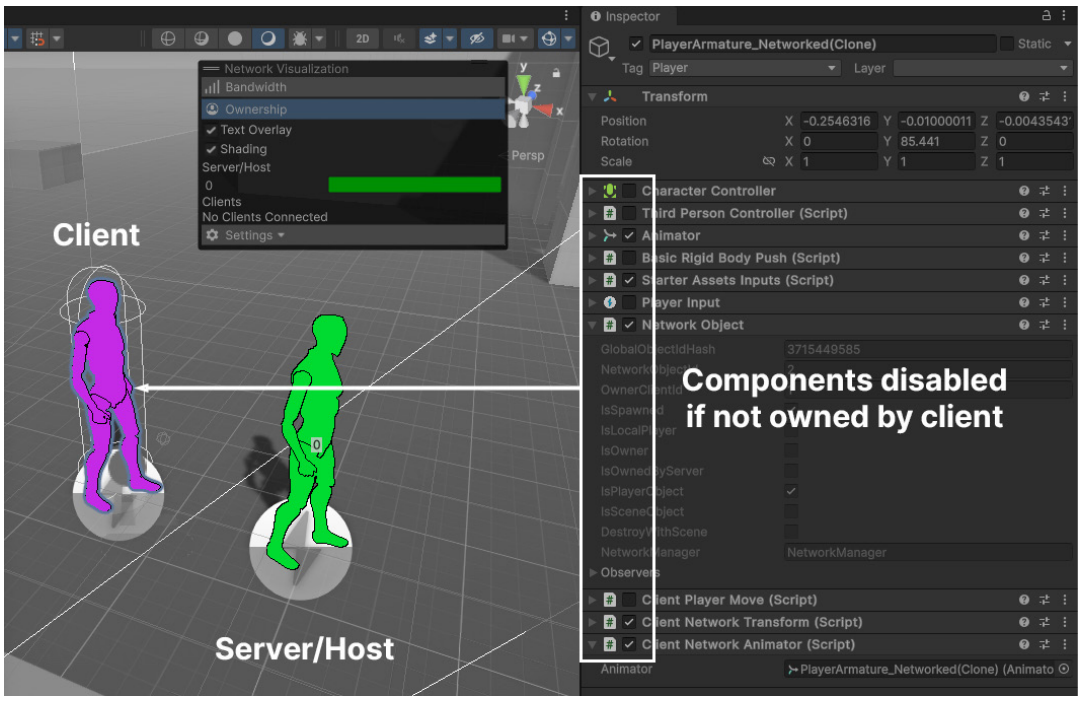
このスクリプトを PlayerArmature_Networked プレハブに加えます。次に、Inspector で適切なフィールドに入力します。



Inspector の ClientPlayerMove フィールドに入力します。

このスクリプトをプレハブに適用したら、ホストセッションとクライアントセッションを接続します。クライアントが NetworkManager に接続すると、ローカルプレイヤーがインスタンスの所有者であるかどうかを確認する `IsOwner` プロパティにより、プレイヤーオブジェクトの特定のコンポーネントがデフォルトで無効になります。

Hierarchy で、PlayerArmature_Networked の 2 つのインスタンスを切り替えて比較します。



所有者でない場合、一部のコンポーネントが無効になります。

PlayerInput などのいくつかのコンポーネントが、それぞれのクライアントが所有していないプレイヤーインスタンスでは、非アクティブになっているのが確認てきます。この設定では、ホストでは 1 つのプレイヤーインスタンスを制御でき、クライアントではもう 1 つのプレイヤーインスタンスを制御できます。

ただし、異なるプレイヤーインスタンスを制御することはできますが、それらの動きはネットワーク間で同期されていません。ホストからクライアントに動きを反映させるには、NetworkTransform などのネットワークコンポーネントを加える必要があります。

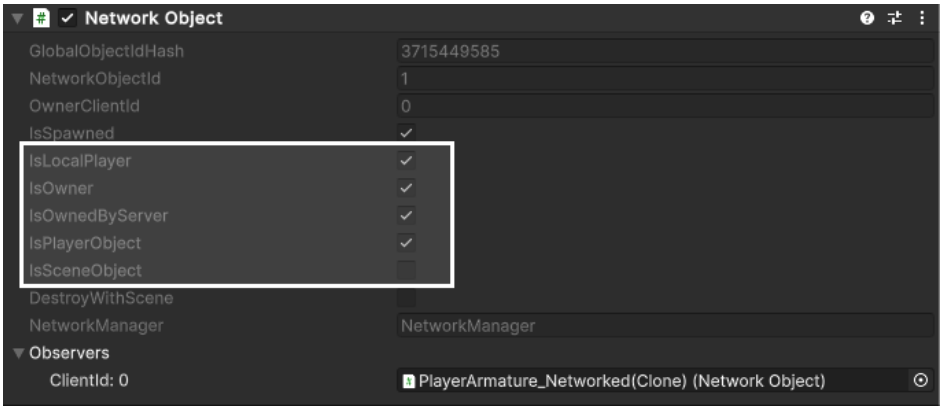
権限と所有権のプロパティ

デフォルトでは、NetworkObject の所有者はサーバーです。ただし、接続済みかつ承認されたクライアントが `SpawnWithOwnership` メソッドを使用することで、NetworkObject を所有することもできます。Netcode for GameObjects はサーバー権威型であるため、サーバーのみが NetworkObject のスポーンとデスポーンを行えます。

NetworkBehaviour には、インスタンスの権限と所有権を簡単に決定する手法がいくつかあります。

- **`IsClient`** は、インスタンスがクライアントで実行されているかどうかを示します。
- **`IsServer`** は、インスタンスがサーバーで実行されているかどうかを示します。
- **`IsHost`** は、インスタンスがホスト (サーバーとクライアントの両方) で実行されているかどうかを示します。
- **`IsLocalPlayer`** は、関連する NetworkObject がローカルプレイヤーオブジェクトかどうかを示します。
- **`IsOwner`** は、オブジェクトがローカルプレイヤーに所有されているか、またはローカルプレイヤーオブジェクト自体であるかを示します。
- **`IsPlayerObject`** は、ゲームオブジェクトが、通常は特定のクライアントによって制御されるネットワークプレイヤーを表しているかどうかを示します。
- **`IsSceneObject`** は、ゲームオブジェクトがデフォルトでシーンに存在しているか、およびゲームプレイ中に動的にスポーンされたオブジェクトではないかどうかを示します。シーンオブジェクトは通常、ネットワーク上で一貫したステートを保持するためにサーバーによって管理されます。

ランタイム時に NetworkObject を調べることで、これらのプロパティの一部を確認できます。



NetworkObject の設定

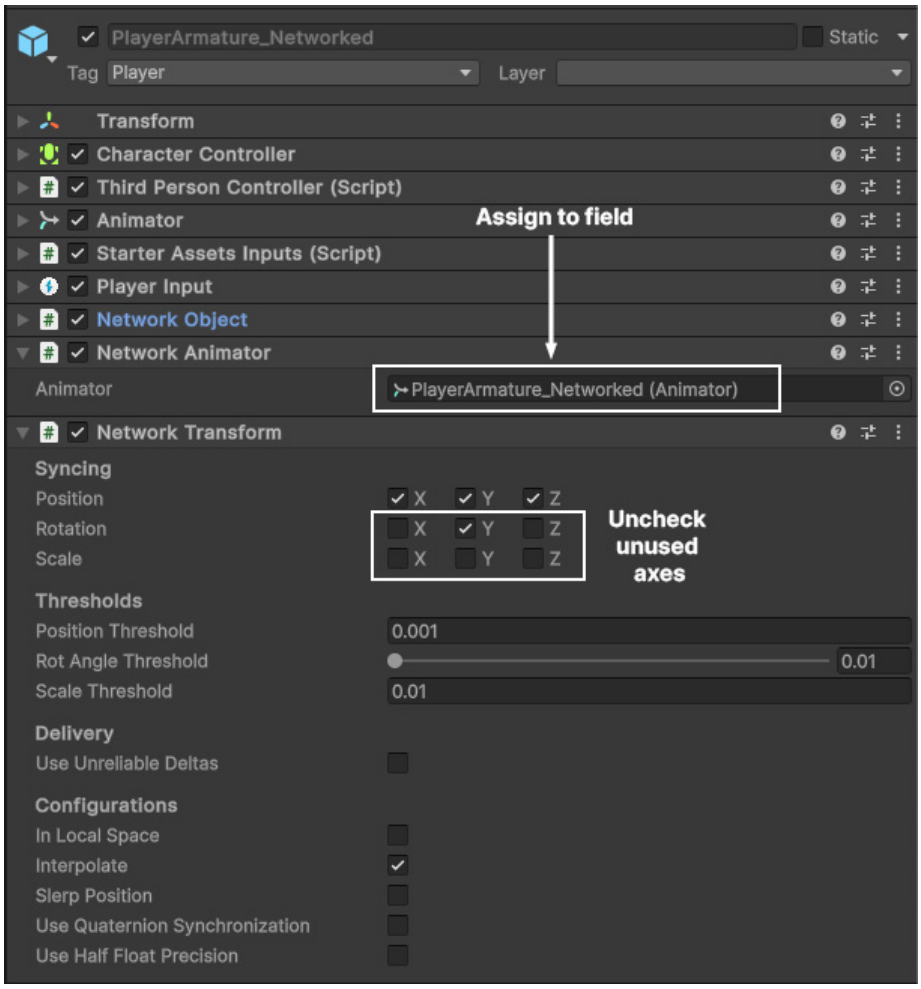
NetworkTransform と NetworkAnimator を使用して同期する

NetworkBehaviour を使用すると、複数のクライアントで同じプレイヤーインスタンスをスポーンできますが、その動きをネットワーク全体で同期させるには、コンポーネントを加える必要があります。

PlayerArmature_Networked プレハブに **NetworkTransform** コンポーネントを加えます。ゲームプレイに影響しない軸はチェックを外します。この例では、Rotation の X 軸と Z 軸だけでなく、Scale のすべてのチェックも外します。同期では帯域幅を使用するため、過剰なデータの同期を最小化することが重要です。

Multiplayer Play Mode では、ホスト側のウィンドウで、コントロールを使用してプレイヤーを動かすと、その動きがクライアント側に同期されることが確認できます。これはネットワークプレイの初歩的な動作を示しています。

次に、PlayerAramature_Networked に **NetworkAnimator** コンポーネントを加えます。空のフィールドに既存の Animator コンポーネントをドラッグアンドドロップします。

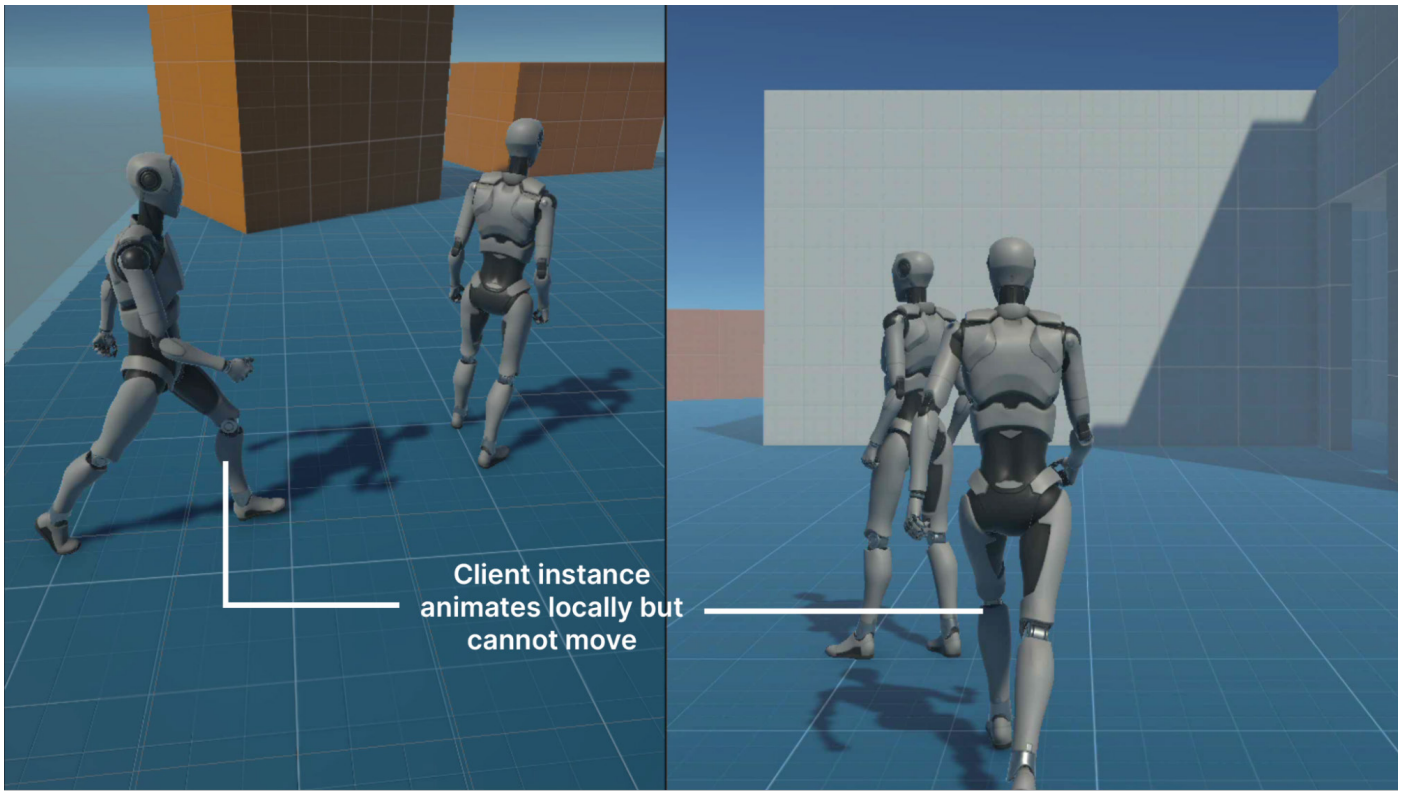


NetworkTransform コンポーネントと NetworkAnimator コンポーネントを加えます。

クライアントウィンドウは、ホストに接続されている別のマシンを表しています。ホストで実行されたアクションがクライアントに反映され、クライアントで実行されたアクションがホストに反映されるのが理想です。

NetworkTransform では、Transform の位置、回転、スケールを同期でき、NetworkAnimator では、アニメーションステートを同期できます。これで、ホストプレイヤーがプレイグラウンド環境内を走ると、その動きが Multiplayer Play Mode のクライアントに転送されます。

ただし、すべてが期待どおりに動作するわけではありません。クライアント側のウィンドウでコントロールを使用してみてください。ホストの動きはクライアントに正しく同期されるにもかかわらず、クライアントの動きはホストに反映されない場合があります。



プレイヤーはその場で走っているように見えます。

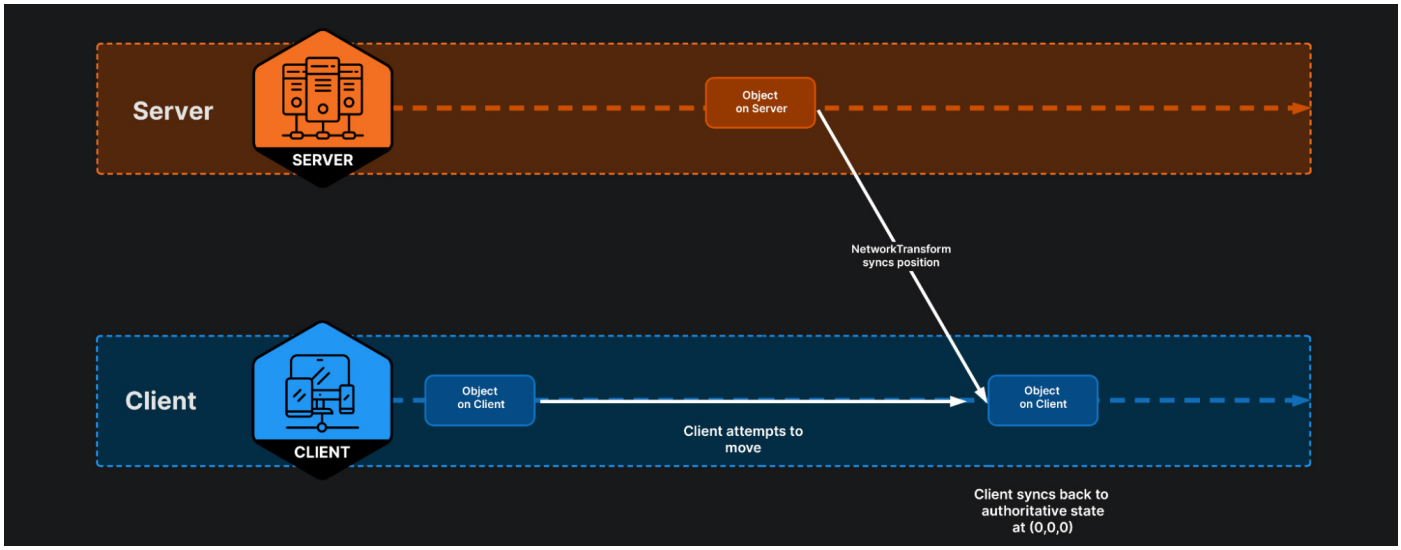
クライアントは入力を受け取り、キャラクターがその場でアニメーションしていることからもそれが確認できますが、プレイヤーインスタンス自体は移動しません。これは、NetworkTransform がサーバー権限で動作し、サーバーの位置情報のみがクライアントに同期されているためです。

クライアントでプレイヤーを移動させようとすると、サーバーがクライアントでの位置をオーバーライドして (0, 0, 0) にリセットします。

クライアント権限を適用する

デフォルトでは、NetworkTransform はサーバー権限モードで動作します。Transform の軸に対する変更はサーバー側で検出され、接続されているクライアントにプッシュされます。

この例では、クライアントでプレイヤーの移動を試みても失敗します。なぜなら、サーバーが (0，0，0) に設定された Transform の権威あるステートを保持しており、それによってクライアント側の変更がオーバーライドされるためです。



サーバー権限はクライアントをオーバーライドします。

これを解決する 1 つの方法は、サーバーからクライアントに権限を移譲することです。そうすることで、クライアントはサーバーによってオーバーライドされることなく、Transform を制御できます。

この動作を実装するには、以下のコード例に示すように **ClientNetworkTransform** コンポーネントを作成し、サーバー権限から所有者権限に切り替えます。

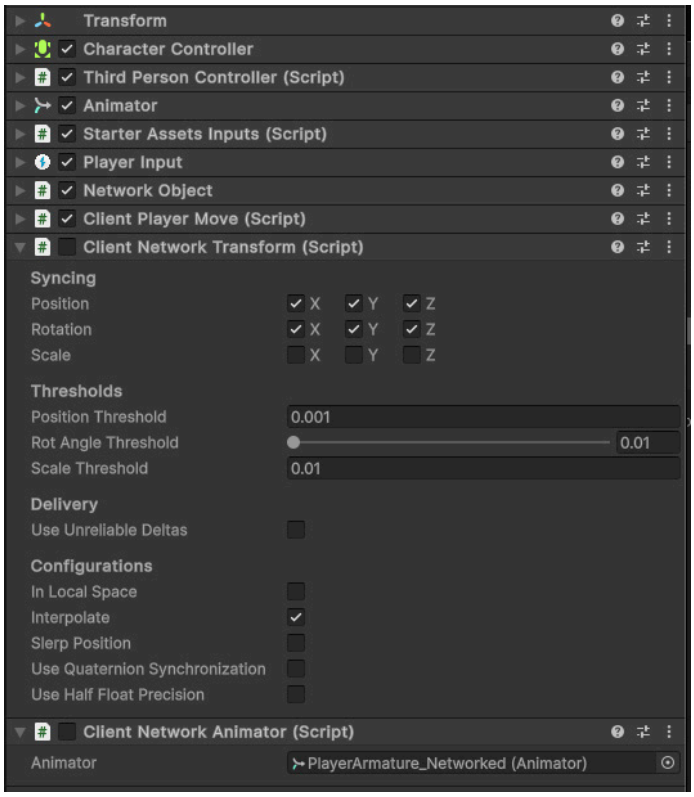
```
using Unity.Netcode.Components;
using UnityEngine;
namespace NetcodeDemo
{
    [DisallowMultipleComponent]
    public class ClientNetworkTransform : NetworkTransform
    {
        protected override bool OnIsServerAuthoritative()
        {
            return false;
        }
    }
}
```

このコードでは、`OnIsServerAuthoritative` メソッドをオーバーライドし、false を返します。プレイヤー用 NetworkObject プレハブで、NetworkTransform をカスタムした ClientNetworkTransform に置き換えます。

同様に、クライアント駆動型の NetworkAnimator を作成することもできます。

```
using Unity.Netcode.Components;
using UnityEngine;
namespace NetcodeDemo
{
    [DisallowMultipleComponent]
    public class ClientNetworkAnimator: NetworkAnimator
    {
        protected override bool OnIsServerAuthoritative()
        {
            return false;
        }
    }
}
```

NetworkAnimator を **ClientNetworkAnimator** に置き換えます。Inspector で Animator フィールドを設定することも忘れないでください。



ClientNetworkTransform と ClientNetworkAnimator を使用します。

Multiplayer Play Mode では、クライアントからプレイヤーの移動が可能になり、位置とアニメーションステートがホストに正しく同期されるようになります。

クライアント駆動の動作は、ネットワークアプリケーションの待ち時間を減らす方法でもあります。" 所有者権威モード " では、ネットワークを介した動作を即座に実行し、高い応答性を発揮します。クライアントは、サーバーとのパケットのラウンドトリップを待つ必要がありません。

ただし、このようなクライアント駆動の動作では、各プレイヤーのユーザー体験を向上させることができる一方で、セキュリティリスクも伴うため、作成時には注意が必要です所有者権威モードでは、アプリケーションの改造やハッキングが可能になるリスクがあります。オンラインの対戦型ゲームでは、機会があればプレイヤーはチートを行うと想定すべきです。これを防ぎ、アプリケーションのセキュリティを強化するには、サーバー権限を選択します。

所有者権威モードのコンポーネント

上記の例のスクリプトは自分で作成することも可能ですが、Unity プロジェクト『*Boss Room*』(`com.unity.multiplayer.samples.coop`) にある Multiplayer Samples Utilities パッケージから、`ClientNetworkTransform` と `ClientNetworkAnimator` のプレビルドコンポーネントを入手することもできます。

この ClientNetworkTransform の実装には、以下のような潜在的な問題があることに注意してください。

- **所有権の移譲:**所有権の切り替えがスムーズに行われない場合があり、オブジェクトが移動したり、同期しなくなったりすることもあります。
- **階層的な所有権:**ClientNetworkTransform は、サーバーが管理する NetworkTransform の子オブジェクトとしてサポートされていません。
- **更新の拒否:**サーバーはクライアントからの更新を拒否できません。なぜなら、このシステムはクライアントとサーバーの共同所有権ではなく、クライアントの所有権のみを認識するからです。
- **インスタンス化した際のオブジェクト移動:**クライアント所有権下でオブジェクトが最初に作成されたとき、サーバーはそのオブジェクトを動かすことはできません。

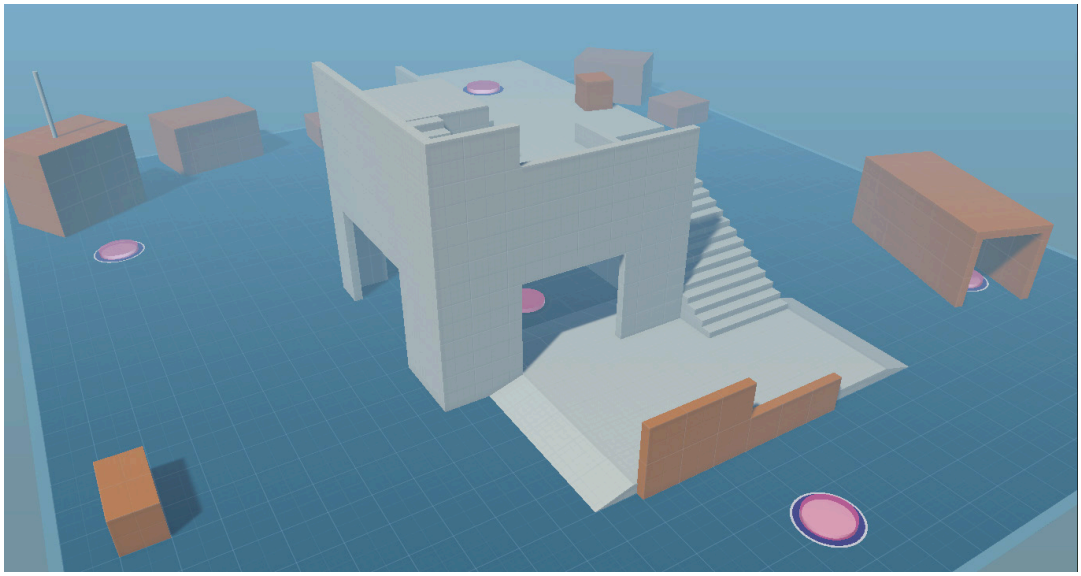
多くの場合、ClientNetworkTransform は、クライアント所有権の Transform をハンドルする際の有効な手段となります。ただし、プロジェクトの一部として実装する前に、これらの制約を考慮する必要があります。

サーバー権限を使用して同期する

応答性の高いゲームプレイを実現するために、クライアントにある程度の権限を与えることも可能ですが、一部の動きはサーバー側でのみ実行できるようにします。一般的に、クライアント制御のアクションから生じる可能性のある、潜在的なバランスの乱れや不公平な利点を回避するために、サーバー権限を使用すべきです。

例えば、クライアントがゲームマップ上のスポーン場所を選択できるようにすると、マップのレイアウトによっては、不当に有利になることがあります。代わりに、事前に定義されたスポーンポイントの 1 セットに対し、サーバーがランダムに割り当てる方が公平です。

これを管理するには、シンプルな見た目のオブジェクトをいくつか定義し、プレイヤーがスポーンする可能性のある場所にそれらのオブジェクトを分散させます。



スポーンポイントは、レベル全体に戦略的に配置されます。

これらのスポーンポイントは。ネットワークを介さない MonoBehaviour で管理できます。ここでは、ServerPlayerSpawnPoints クラスが m_SpawnPoints というリストを持ち、各スポーンポイントのゲームオブジェクトを参照します。

このサンプル実装では、Unity が作成した Asset Store プロジェクト *ゲームプログラミングのパターンと SOLID を活用してコードをレベルアップさせる* から流用した、汎用的なシングルトンパターンを使用しています。

```
public class ServerPlayerSpawnPoints : Singleton<ServerPlayerSpawnPoints>
{
   [SerializeField]
   private List<GameObject> m_SpawnPoints;
   public GameObject GetRandomSpawnPoint()
   {
       if (m_SpawnPoints.Count == 0)
           return null;
       return m_SpawnPoints[Random.Range(0, m_SpawnPoints.Count)];
   }
}
```

ServerPlayerMove という名前の NetworkBehaviour は、ServerPlayerSpawnPoints のインスタンスを使用してスポーンポイントをランダムに選択します。

```
using Unity.Netcode;
using UnityEngine;
[DefaultExecutionOrder(0)] // ClientNetworkTransform より先に実行する
public class ServerPlayerMove : NetworkBehaviour
{
    public override void OnNetworkSpawn()
    {
        // サーバーでのみ実行する
        if (!IsServer)
        {
            enabled = false;
            return;
        }
        SpawnPlayer();
        base.OnNetworkSpawn();
    }

    // スポーン時に利用可能な次のポイントに移動させる
    void SpawnPlayer()
    {
        var spawnPoint = ServerPlayerSpawnPoints.Instance.GetRandomSpawnPoint();
        var spawnPosition = spawnPoint ? spawnPoint.transform.position : Vector3.zero;
        transform.position = spawnPosition;
    }
}
```

すべてのロジックは OnNetworkSpawn で行われます。クライアントが接続するたびに SpawnPlayer を呼び出し、プレイヤーをランダムに選択されたスポーンポイントから開始させます。IsServer のチェックにより、これがサーバーでのみ発生することが保証され、権威あるゲームステートが維持されます。

ServerPlayerMove スクリプトを PlayerArmature_Networked プレハブに加えます。Multiplayer Play Mode を開始すると、各クライアントがプレイグラウンド環境内のランダムなポイントに接続され、スポーンされます。

この実装は、NetworkBehaviour がネットワーク制御されていないシーン内の要素と相互作用できることを示しています。ここでは、Hierarchy にすでに設定されている静的データとシーンオブジェクトを活用します。

クライアントが接続すると、ServerPlayerMove は既存のゲームプレイシーンからランダムに 1 つのスポーンポイントを取得するだけで済みます。これにより、ネットワーク経由で伝送されるデータの量を抑えることができます。

プレイヤーはランダムなスポーンポイントに出現します。

重要なポイント：

— ClientNetworkTransform は所有者権限であるため、Awake 中は CharacterController コンポーネントを無効にすることが重要です。`ServerPlayerMove` によってプレイヤーが配置されたら、CharacterController を再度有効にします。これにより、計算値がオーバーライドされて、ゲームワールドの中心位置にリセットされないようにします。

— 同様に、Inspector で `m_SpawnPoints` を入力して、プレイヤーが (0,0,0) にスポーンされないようにします。

— `DefaultExecutionOrder` 属性を小さい値に設定して、`ServerPlayerMove` が `ClientPlayerMove` より前に実行されるようにします。 例えば、`[DefaultExecutionOrder(0)]` を使用すると、`ServerPlayerMove` の優先度が高くなり、最初に実行できるようになります。

これで、マルチプレイヤープロジェクトで複数のクライアントを 1 つのホストに接続できるようになりました。ゲーム内では、三人称視点のプレイヤーキャラクターをレベル内の指定された位置にスポーンし、その動きとアニメーションをリアルタイムで同期させることができます。

この同期はマルチプレイヤー体験に不可欠です。NetworkTransform や NetworkAnimator などのコンポーネントにより、特別な作業なしにこのプロセスを進めることができますが、ゲームプレイでは、独自の NetworkBehaviour をカスタマイズする必要があります。

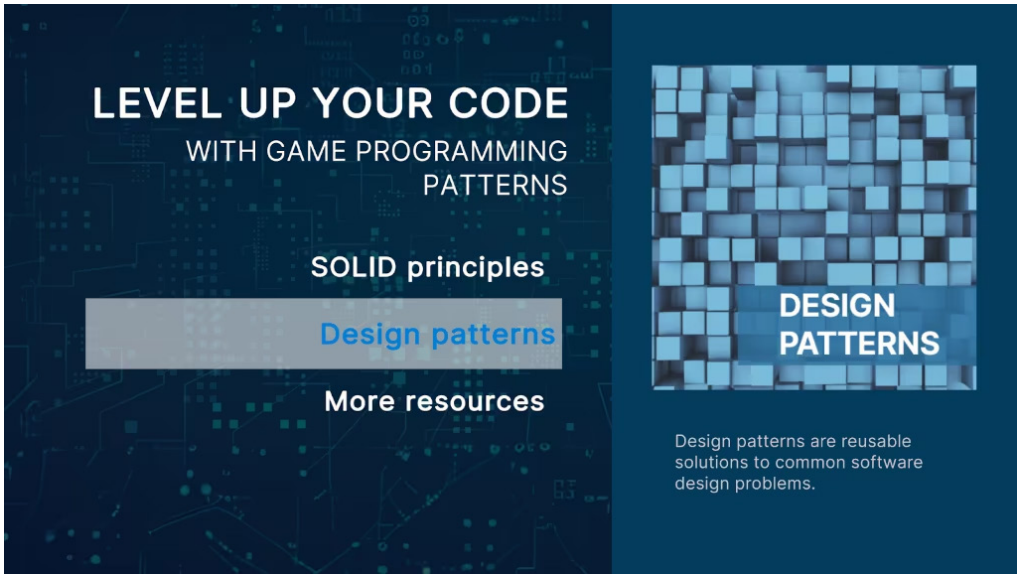
次は、ネットワークワークを介してデータとゲームステートを同期する別の方法を学んでいきます。

シングルトンデザインパターン

シングルトンとは、ランタイム時に特定の型に属する一意のインスタンスにアクセスできる便利な仕組みです。ただし、シングルトンは依存関係が強くなる可能性があるため、その欠点に注意してください。

Netcode for GameObjects では、NetworkManager.Singleton を参照するたびにシングルトンを使用します。サンプルプロジェクトには、あらゆる種類の MonoBehaviour で使用できる汎用的なシングルトンの例も含まれています。

シングルトンの詳細については、eBook『*ゲームプログラミングのパターンと SOLID を活用してコードをレベルアップさせる*』を参照してください。このガイドブックでは、シーン内のオブジェクトの通信に活用できる、イベントやイベントチャンネルといったシングルトンとは別のパターンを紹介しています。



Unity の無料 eBook で、設計パターンについて学びましょう。プログラマー、テクニカルアーティスト、アーティスト、デザイナー向けのあらゆる上級者向けガイドについては、Unity のベストプラクティスを参照してください。

ネットワーク同期

ネットワーク同期は、すべてのプレイヤーにとって一貫性があり、公平なゲーム体験を維持するために不可欠です。

これまで、クライアント駆動のモデルを使用してプレイヤーの動きとアニメーションをプレイヤー用 NetworkObject と同期する方法を説明してきました。しかし、多くの場合、ゲームプレイにはプレイヤーキャラクター以外の要素も関与します。ゲームデザインによっては、プレイヤーが投射物を撃つ、ドアを開ける、または他のシーンオブジェクトと相互作用する必要があります。これらのインタラクションは、ネットワーク接続可能であるとともに、独自のゲームステートをクライアントとサーバー間で同期する必要があります。

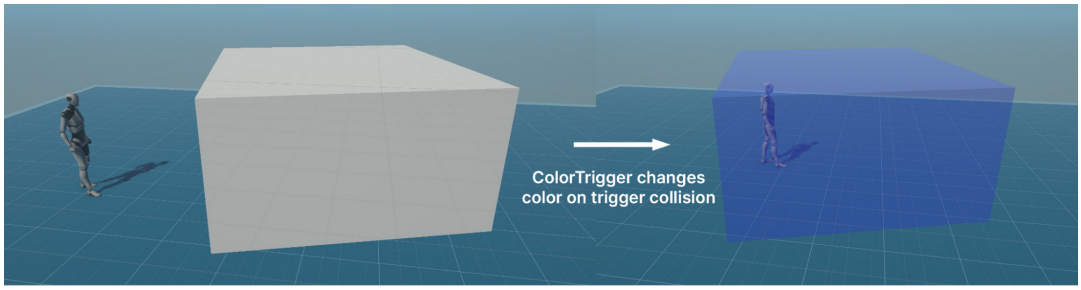
このセクションでは、プレイヤーがゲーム環境の一部で相互作用する場合があるゲームプレイアクション用に、クライアントとサーバー間の通信を設定します。これには、ネットワーク接続されたゲームステートの実装と、サーバーとリモートプロシージャコール (RPC) の送受信について取り上げます。

ゲームプレイメカニクス

サーバーとクライアント間の通信について説明するため、簡単なゲームメカニクスを再現してみましょう。

まず、プレイヤーに接触すると色が変わるトリガーコライダーをゲームレベル内に加えます。あるプレイヤーがタッチすると 1 つの色に、別のプレイヤーがタッチすると別の色に変化するようにします。

シーンで、新しいゲームオブジェクト (例 : キューブなど) を作成します。BoxCollider コンポーネントを加え、IsTrigger オプションを有効にします。透明なマテリアルを作成し、MeshRenderer に割り当てます (この例では URP/Lit シェーダーを使用します)。初期色を白などニュートラルなものに設定します。



このトリガーは、ネットワークを介して色の値を受け取ります。

次に、ネットワーク同期を加える必要があります。NetworkVariable と RPC を使用して、色の変更をネットワーク上で同期させましょう。

NetworkVariable を定義する

NetworkVariable は、ネットワーク上でステートを同期管理するために設計された特殊な変数です。NetworkVariable に対するサーバー上での変更は、すべてのクライアントに反映されます。これは、位置や HP など、継続的に同期されるデータに最適です。今回のケースでは、トリガーの色ステートを同期するために使用します。

`ColorTrigger` という名前の NetworkBehaviour を作成し、トリガーオブジェクトにアタッチします。このスクリプトには、`Color` 値を含む `m_NetworkColor` という NetworkVariable を加えます。

```
using UnityEngine;
using Unity.Netcode;
public class ColorTrigger : NetworkBehaviour
{
   public NetworkVariable<Color> m_NetworkColor = new NetworkVariable<Color>(Color.
white);
   private Material m_InstanceMaterial;

   public override void OnNetworkSpawn()
   {
       m_NetworkColor.OnValueChanged += OnColorChanged;
       MeshRenderer meshRenderer = GetComponent<MeshRenderer>();
       if (meshRenderer != null)
       {
           m_InstanceMaterial = new Material(meshRenderer.material);
           meshRenderer.material = m_InstanceMaterial;
           UpdateMaterialColor(m_NetworkColor.Value);
       }
   }
```

```
    public override void OnNetworkDespawn()
    {
        m_NetworkColor.OnValueChanged -= OnColorChanged;
    }
    private void OnColorChanged(Color oldColor, Color newColor)
    {
        UpdateMaterialColor(newColor);
    }

    private void UpdateMaterialColor(Color newColor)
    {
        if (m_InstanceMaterial != null)
        {
            m_InstanceMaterial.SetColor("_BaseColor", newColor);
        }
    }
}
```

プレイヤーがトリガーを入力すると、このスクリプトはマテリアルインスタンスの通常色プロパティを切り替えます。スクリプトでは、`m_NetworkColor` という NetworkVariable を使用します。

この仕組みの詳細は以下のとおりです。

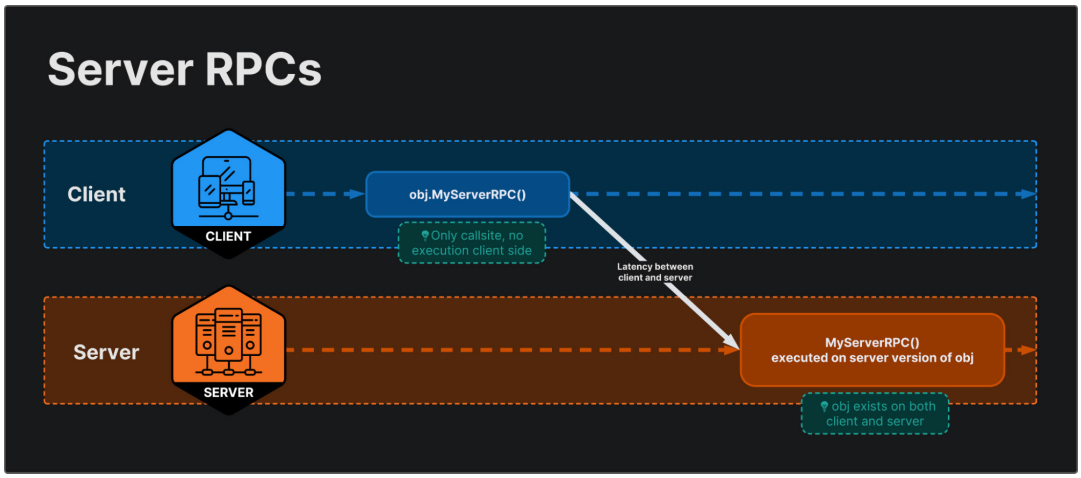
- この NetworkVariable は実際の色の値を追跡し、すべてのクライアント間で同期します。スクリプトはサーバーとクライアントの両方で実行されますが、デフォルトでは、NetworkVariable への書き込み権限があるのはサーバーのみです。クライアントは、その `Color` の値を読み取ることのみが可能です。
- `OnValueChanged` イベントは、NetworkVariable が変更されるたびに、トリガーのマテリアル通常色を更新します。スクリプトは `OnNetworkSpawn` でイベントに登録し、`OnNetworkDespawn` で登録を解除します。

NetworkTransform は、Transform データ (位置、回転、スケール) の同期に特化していますが、NetworkVariable は、プリミティブ型、カスタム構造体、ゲームステート管理に必要なその他のデータなど、より一般的なデータ型を同期できます。

クライアントで作業する場合、NetworkVariable はサーバー権限のため、直接変更することはできません。代わりに、クライアントはすべての変更をサーバーに通知する必要があります。サーバーはステートを更新して再度反映し、その変更が有効になった時点でクライアントに表示されます。

このような変更の通信をハンドルするには、RPC を使用します。RPC を使用すると、あるデバイスが別のデバイスに対して、特定のアクションや更新を実行するように要求できます。RPC はクライアントからサーバーに対して呼び出すことも、その逆を行うことも可能です。

これを実現するために RPC を実装する方法を見てみましょう。



サーバー RPC は、クライアントからサーバーに対してリモートで実行されます。

RPC を加える

RPC を使用すると、サーバーまたは他のクライアント上で関数をリモートで呼び出すことができます。[Rpc(SendTo.Server)] とマークされたメソッドは、クライアントからサーバーに呼び出され、[Rpc(SendTo.Client)] とマークされたメソッドは、サーバーからクライアントに呼び出されます。なお、従来の構文である [ServerRpc] や [ClientRpc] を使用して、サーバー RPC またはクライアント RPC を指定することもできます。

RPC は他のメソッドとよく似ていますが、いくつかの規則に従う必要があります。

- **Rpc 属性 :** メソッドに [Rpc] 属性を付け、指定可能なターゲットを指定します (例 : [Rpc(SendTo.Server)] はサーバーでのみメソッドを呼び出します)。
- **命名規則 :** メソッド名の末尾には、サフィックス “Rpc” を付けます (例 : DoSomethingRpc)。

RPC は、プレイヤーアクションや特定のゲームステートの変更など、継続的な同期を必要としない個別のイベントに適しています。

TriggerColor スクリプトに以下のメソッドを加えます。

```
private void OnTriggerEnter(Collider other)
 {
     NetworkObject networkObject = other.GetComponent<NetworkObject>();
     if (IsClient && networkObject != null && networkObject.IsOwner)
     {
         ChangeColorServerRpc(networkObject.OwnerClientId);
     }
 }
```

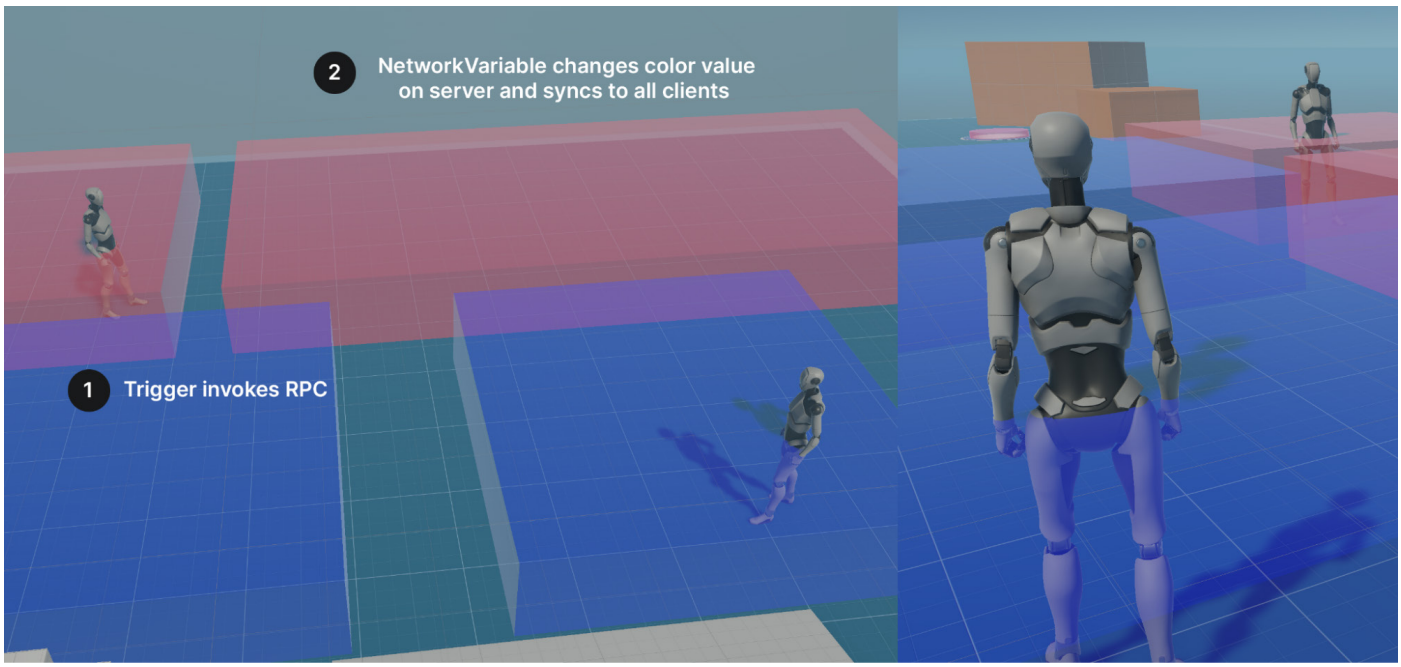
```
  [Rpc(SendTo.Server)]
private void ChangeColorServerRpc(ulong playerId)
{
    // シンプルなチームシステム: 偶数の場合は青、奇数の場合は赤
    Color newColor =
        (playerId % 2 == 0) ? new Color(0, 0, 1, 0.5f) : new Color(1, 0, 0, 0.5f);

    m_NetworkColor.Value = newColor;
}
```

シングルプレイヤーゲームでは、OnTriggerEnter の中に適切なロジックを記述できます。ただし、マルチプレイヤーゲームでは通常、すべてのクライアントの同期を維持するために、クライアントとサーバー間の通信を使用してインタラクションをハンドルします。

m_NetworkColor の値を直接設定する代わりに、OnTriggerEnter で現在のゲームインスタンスがクライアントかどうかを確認します。クライアントである場合は、ChangeColorServerRpc を呼び出し、OwnerClientId を渡します。

このメソッドでは、プレイヤーの ID に基づいて新しい色を決定し (この例では、ID が偶数なら青、奇数なら赤)、m_NetworkColor の値を更新します。

この変更は、接続されているすべてのクライアントに伝播され、すべてのゲームインスタンスで一貫したゲームステートビューが確保されます。m_NetworkColor が変更されると、すべてのクライアントで OnColorChanged メソッドがトリガーされ、トリガーのマテリアルの色が更新されます。



このシンプルなゲームメカニクスは、クライアントとサーバーのインタラクションを示しています。

トリガーメカニクス

これはシンプルな例ですが、似たようなゲームメカニクス (プレイヤーのアクションをトリガーにして環境が視覚的に変化するような仕組み) は、多くのマルチプレイヤーゲームで使用されています。例：

- 協力型パズルゲームでは、プレイヤーがボタンやトリガーを使用して環境と相互作用する必要があります。ゲーム環境で発生する視覚的なサインは、全プレイヤーにとって、アクションを調整するための信号として機能します。
- 対戦型シューティングゲームでは、プレイヤーはレベル上のコントロールポイントを制圧することができます。コントロールポイントの色が管理するチームによって変わる仕組みは多く使用されています。これも、プレイヤーが戦略を調整するための視覚的な手がかりになります。
- オンライン RPG では、レベルの特定のエリアに入ると、バフやデバフなどの効果をトリガーできます。

RPC と NetworkVariable の比較

データを同期する際、NetworkVariable と RPC のどちらを選択するかはユースケースによって異なります。

- NetworkVariable と NetworkTransform は、位置や HP など、継続的に同期されるデータに最適です。今回のケースでは、トリガーの色ステートを同期するために使用します。この例では、ColorTrigger が アクティブな色を格納するために、NetworkVariable を使用しています。
    - ユースケース:HP、位置データ、ゲームスコア
- リモートプロシージャコール (RPC) は、プレイヤーアクションや特定のゲームステートの変更など、継続的な同期を必要としない個別のイベントに適しています。この例では、プレイヤーがトリガーゾーンに入った際に RPC がすべてのクライアントに通知し、プレイヤーの ID に基づいて色を変更するようサーバーに求めます。
    - ユースケース:プレイヤーアクション (例: 射撃、スキル使用)、ゲームイベント (例: スポーン、ゲーム開始、ゲームクリア)

どちらのメカニズムもネットワークを介したゲームでのデータ同期には不可欠で、NetworkVariable は継続的なステート管理、RPC は特定のイベントおよびアクションのハンドルに使用されます。

こうしたゲームプレイの例を参考にすると、NetworkVariable と RPC のどちらを使用すべきか判断しやすくなります。

タスク/システム	RPC	NetworkVariable
インベントリ管理	アイテムがピックアップまたは使用された際にクライアントに通知し、インベントリに対して追加または削除します。	各プレイヤーの現在のインベントリを維持します。インベントリは、アイテムのリストを同期する NetworkList として実装できます。
戦闘システム	攻撃や必殺技などの戦闘アクションを実行します。RPC を使用すると、ダメージや効果を適用できます。	各プレイヤーの HP や状態を追跡します。HP や発動中の状態効果は、NetworkVariable を使用して同期できます。
環境とのインタラクション	ドアを開ける、メカニズムを起動するといった特定のアクションをトリガーします。	相互作用するオブジェクトのステート (ドアが開いているか閉じているかなど) を維持します。
目標	目標の達成を通知します。	目標に向かって進行中の進捗を追跡します。収集アイテムや完了したタスクの数を NetworkVariable に格納します。

詳細については、RPC と NetworkVariable の比較に関するドキュメントページを参照してください。

マルチプレイヤー向けの設計

これで、ネットコードの基本に慣れました。ゲームを作成する際には、"ネットワークを介したマルチプレイヤー" という考え方を取り入れることが不可欠です。シングルプレイヤーゲームではシンプルだったことが、複数のデバイスで同じ動作を実現しようとすると、NetworkVariable や RPC を組み込む必要が生じて複雑になることがよくあります。

クライアントとサーバー間でステートを同期する方法を計画しましょう。継続的に同期するデータには NetworkVariable を使用し、一時的なイベントには RPC を使用します。次に、必要なものだけを同期して、ネットワークトラフィックを最小化します。

シングルプレイヤーゲームをマルチプレイヤーゲームに変換することも可能ですが、通常は開発開始時からマルチプレイヤーを念頭に置いて設計した方が効率的です。オブジェクトとアクションについて、どれをサーバーが所有し、どれをクライアントが管理できるかを早期の段階で決めておきます。サーバー権限は、セキュリティと一貫性を最大限に確保しますが、特定のアクションの待ち時間を減らすために、クライアント権限とのバランスを取りましょう。

この待ち時間は、ネットワークを介したゲームを構築するうえで足かせとなる可能性があります。次は、この待ち時間がマルチプレイヤー体験にどのような影響を与えるかを見ていきます。

開発のヒントとして、こちらの Unite 2024 セッション(YouTube 自動翻訳字幕推奨) を参照してください。このセッションでは、マルチプレイヤーゲーム開発において最もよくある重大なミスと、それらを回避して、成功率を高める方法を紹介しています。

ネットワーク待ち時間とパフォーマンス

オンラインゲームをプレイしたことがある方であれば、待ち時間によってゲーム体験がどれほど台無しになるのかを直接実感したことがあると思います。帯域幅が狭く、ネットワーク接続が不安定だと、プレイヤーの動きがカクついたり、フレームレートが不安定になったり、入力ラグが目立ったりします。

クライアントとサーバー間の通信方法を理解した今なら、この現象が起こる理由も理解できるでしょう。インターネットが距離の離れたデバイス同時をつないでいる状況では、多くの問題が発生する可能性があります。Unity Transport の信頼性が高くても、UDP パケットが宛先に到達するまでに失われたり、順序通りに届かなかったり、破損したりすることがあります。これらはすべて、待ち時間の原因になる可能性があり、ラグとしても認識されます。

待ち時間のシミュレーション

開発中に待ち時間の影響を把握して軽減するには、UnityTransport の Debug Simulator または Network Simulator を使用して、Unity エディターでさまざまなネットワーク状態をシミュレートします。

Unity Transport の Debug Simulator

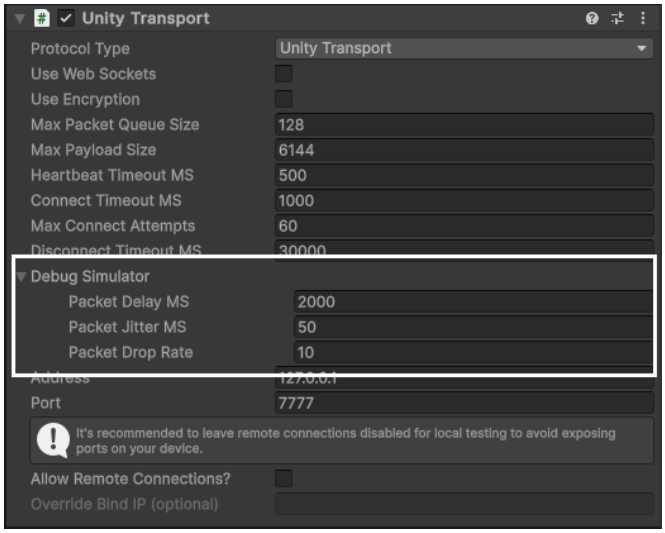
UnityTransport コンポーネントにある Debug Simulator を調整して、待ち時間、ジッター、パケットロスを人為的に発生させます。これらの設定は、シーンで NetworkTransport オブジェクトを選択した際に、Inspector に表示されます。いくつかの値を以下のように設定することで、ネットワーク輻輳を再現できます。

待ち時間：ミリ秒単位で遅延を加えて、低速のネットワーク接続をシミュレートします。例えば、待ち時間を 100 ミリ秒に設定すると、中程度のネットワーク遅延を模倣できます。

ジッター：待ち時間にばらつきを加えることで、不安定なネットワーク状態をシミュレートします。例えば、ジッターを 50 ミリ秒に設定すると、接続の不安定さがゲームプレイに与える影響を確認できます。

パケットロス：ドロップするパケットの割合を指定して、低品質の接続を模倣します。パケットロスを 5% に設定して、ゲームプレイへの影響を確認してみましょう。

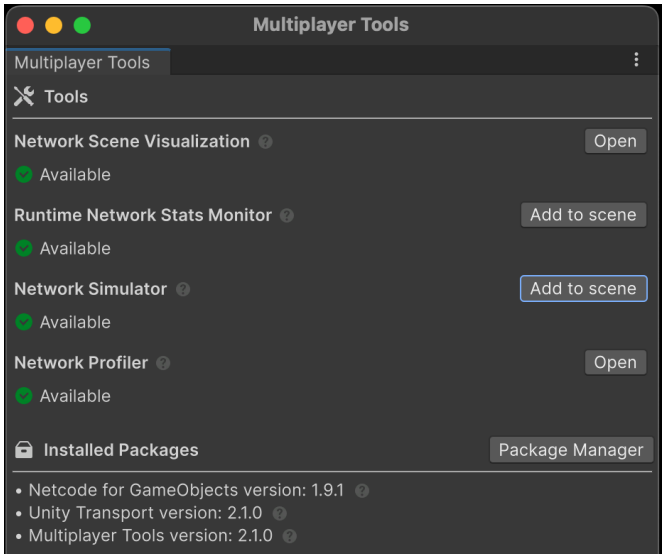
ゲームアプリケーションをもう一度再生して、こうしたシミュレーション条件がゲームプレイにどのような影響を与えるのかを確認します。



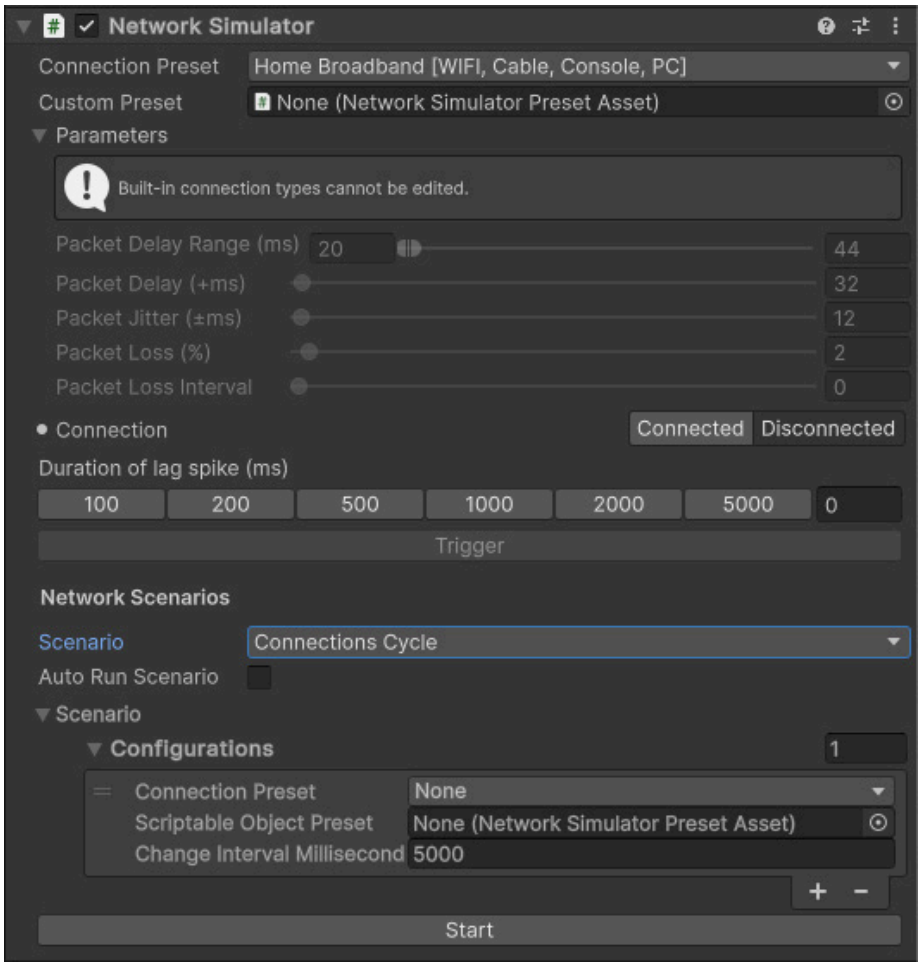
Unity Transport の Debug Simulator を調整します。

Network Simulator

Multiplayer Tools パッケージに含まれる Network Simulator を使用すると、理想的でないネットワーク状態をテストできます。これにより、本番環境で問題が表面化する前に、検出して修正することができます。ネットワークの接続切断、ラグスパイク、パケットロスなどのネットワークイベントのシミュレーションを簡単に行えます。



Multiplayer Tools パッケージから Network Simulator をインストールします。



Network Simulator で待ち時間をテストします。

その他のネットワークコンディショナー

Debug Simulator または Network Simulator は、エディター上でのみ機能します。ランタイムビルドで待ち時間をシミュレートするには、別のネットワークコンディショナーを使用します。Windows 用の clumsy や、macOS/iOS 用の Network Link Conditioner などのツールを使用すると、さまざまなネットワーク状態を再現して徹底的にテストできます。

詳細については、このガイドの後半のテストとデバッグセクションを参照してください。

待ち時間がアプリケーションパフォーマンスに与える影響に対処することは、マルチプレイヤー開発における最大の課題の 1 つです。幸いなことに、待ち時間の影響を軽減するのに役立つ戦略がいくつかあります。そのうちの一部を見ていきましょう。

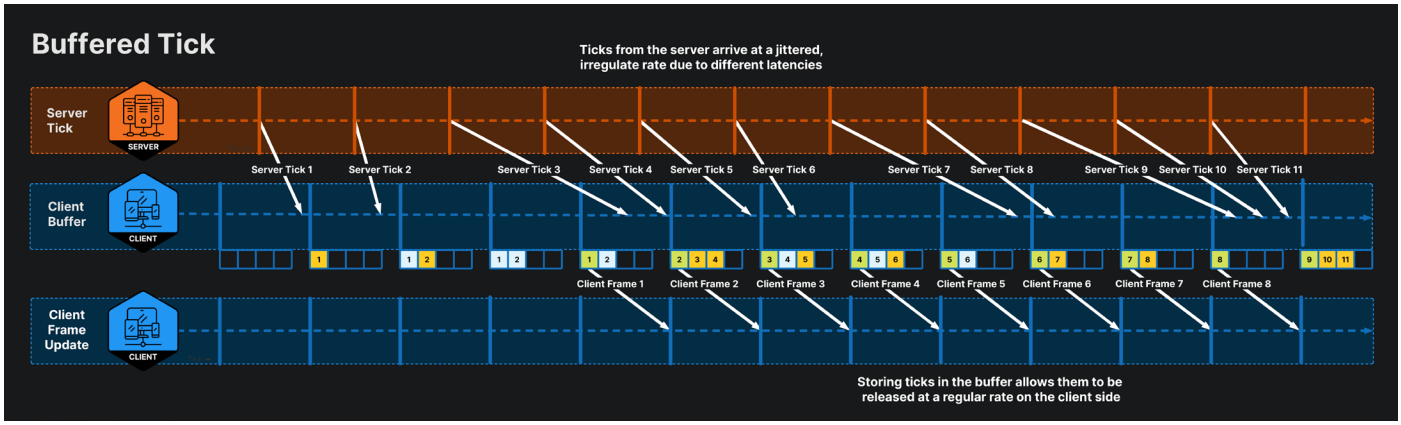
クライアントサイド補間

待ち時間の影響を軽減する方法の 1 つが、クライアントサイド補間です。この方法では、各クライアントは即座にレンダリングせず、意図的に短い補間期間を設けてレンダリングを遅らせます。

クライアントサーバー型トポロジーでは、クライアントは通常、サーバーよりラウンドトリップタイム (RTT) の約半分遅れてステートをレンダリングします。クライアントサイド補間では、さらに意図的な遅延を加えます。

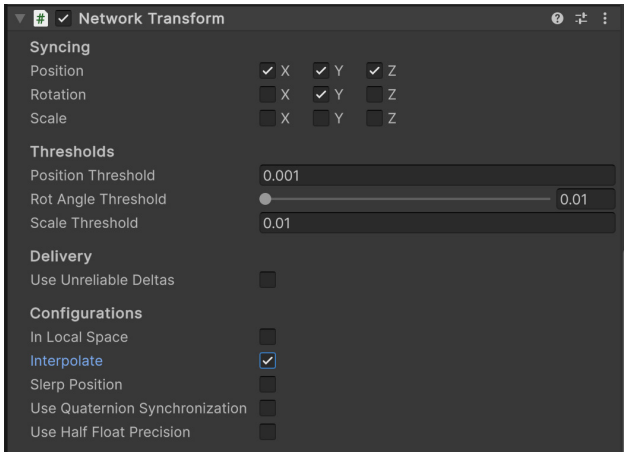
少し遅れて実行することで、クライアントはサーバーか受け取るステート更新にバッファを持たせることができます。 次の更新をレンダリングするとき、クライアントは直近の 2 つのサーバーティックから、補間されたステートを計算します。

バッファを使用すると、サーバーからのティックが不安定かつ不規則な間隔で到着した場合でも、クライアントは定期的なクライアント更新をレンダリングできます。この補間されたステートにより、わずかな待ち時間やジッターが隠されます。



クライアントは、バッファから補間されたステートをレンダリングします。

クライアントサイド補間は Netcode for GameObjects で NetworkTransform コンポーネントのフラグとして使用できます。**Interpolation** を有効にすると、関連するゲームオブジェクトの位置、回転、スケールが補間されます。



NetworkTransform でクライアントサイド補間が有効になっています。

クライアントサイド補間では、レンダリングにわずかな遅延が生じることに注意してください。これは補間に不可欠なものです。

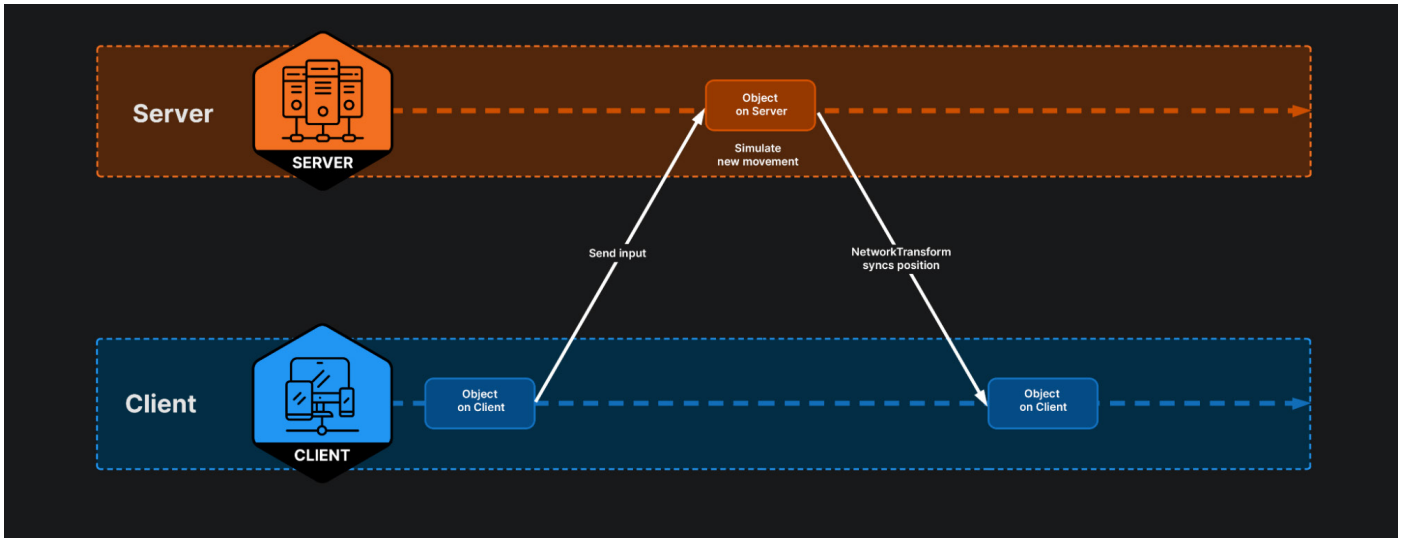
クライアントサイド予測と先読み

先ほどの例では、`ClientNetworkTransform` によって、クライアントにプレイヤーの動きを直接制御させることで、ゲームの応答性と即時性を高めていました。しかし、多くのゲームでは、ゲームデザイン、公平性、セキュリティといったいくつかの要因により、クライアントに権限を持たせることが不可能です。

サーバー権限が必要な理由

公正な競争が不可欠な対戦型ゲームでは、クライアントに権限を持たせると、プレイヤーがクライアント側のコードやデータを改ざんし、ゲームに対するチートや悪用が行えるようになります。プレイヤーインタラクションが複雑なゲームや、ゲームワールドを共有するゲームでは多くの場合、データの整合性を確保し、改ざんを防ぎ、すべてのプレイヤーで一貫した体験を維持するためのサーバー権限が求められます。このような理由から、公平性を確保し、ゲームの整合性を維持するにはサーバー権限が不可欠です。

そのため、所有者権威型の `ClientNetworkTransform` コンポーネントではなく、標準の `NetworkTransform` コンポーネントを使用する必要があります。これにより、クライアント入力がプレイヤーを制御することがなくなり、サーバーのみが制御できるようになります。



サーバー権限を使用して NetworkTransform を移動させています。

ただし、サーバー権限が原因で待ち時間が長くなることがあります。各クライアントは、画面上のプレイヤーを直接操作するのではなく、その入力をサーバーに送信する必要があります。その後、サーバーは受け取った入力を処理してゲームをシミュレートし、ゲームステートを計算します。クライアントがシミュレーションの結果を受信したときにのみ、次のフレームをが表示されます。

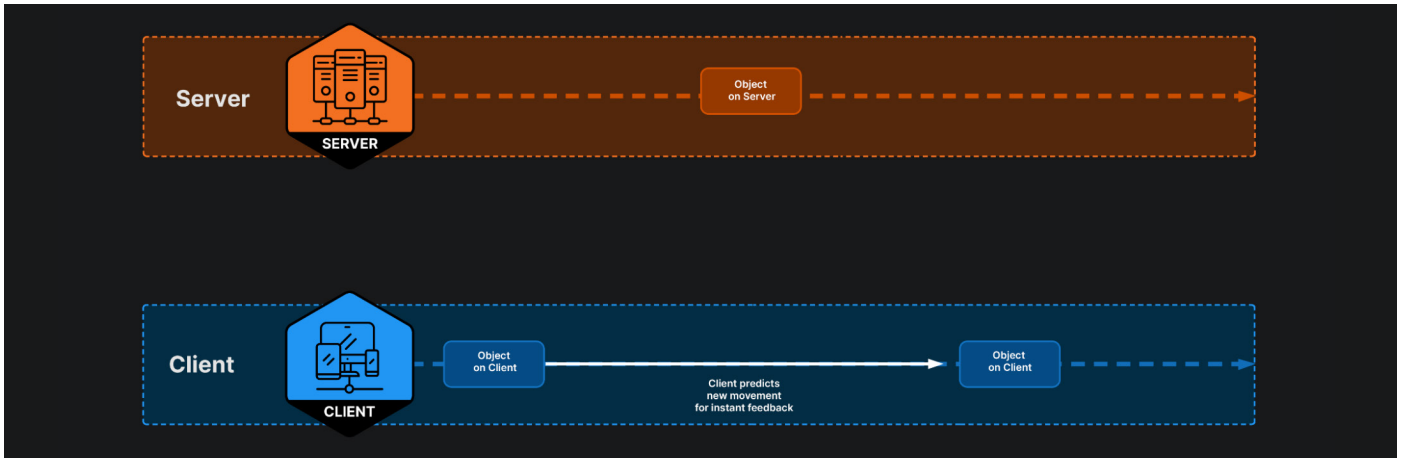
権威サーバーを使用している際にプレイヤーが待ち時間に気づくのは、データがクライアントからサーバーに移動したり戻ったりするのに時間がかかるためです。このシナリオでは、UDP パケットがインターネットを経由する必要があるため、クライアントはサーバーよりも遅れる傾向があります。

ゲーム内の多くの要素に関しては、このラグは許容範囲かもしれませんが、プレイヤーキャラクターなどの一部の要素においては、このようなラグはゲーム体験を損ない、プレイを困難にすることがあります。

クライアントサイド予測の仕組み

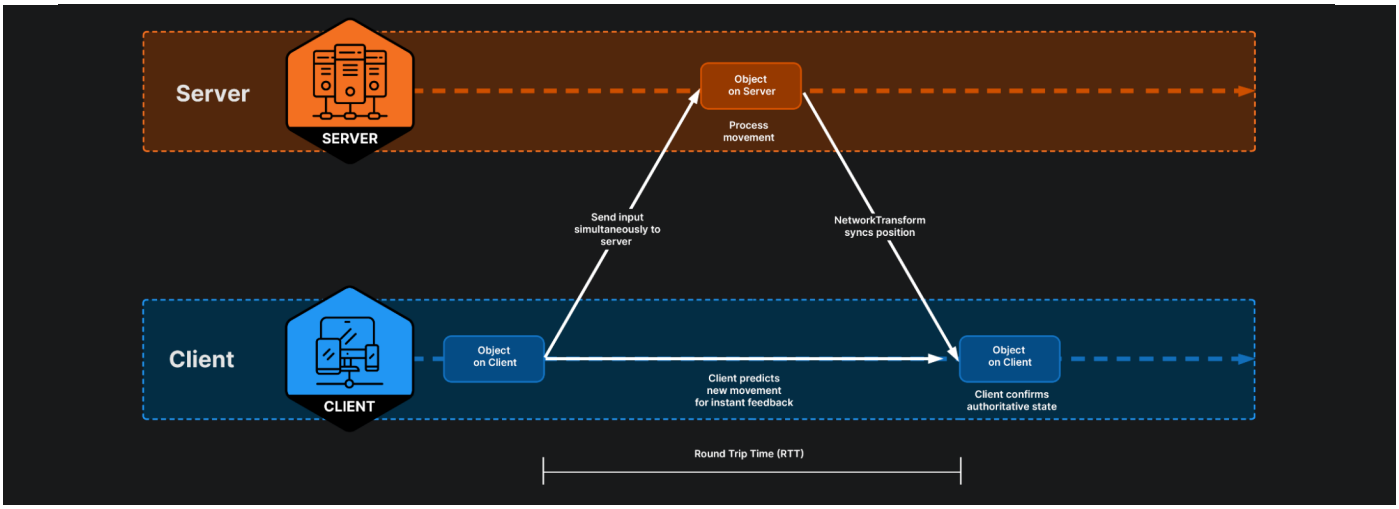
クライアントサイド予測は、サーバー権限によって生じるラグに対する 1 つの解決策を提供します。サーバーからの応答を待つ代わりに、クライアントはほんの一瞬先のゲームステートを予測し、ゲームを視覚的に更新します。これが " 予測 " と呼ばれるのは、クライアントがサーバーの実際のステートを知らない段階でゲームステートを予測するためです。

この方法により、クライアントはプレイヤーのアクションに対して視覚的なフィードバックを瞬時に返し、ゲームの応答性を高めることができます。



クライアントがゲームステートを予測します。

同時に、クライアントはプレイヤーのアクションをパケットとしてサーバーに送信します。サーバーはクライアントの入力パケットを受け取り、その入力を使用してゲームステートをシミュレートします。サーバーは受け取った入力を処理してゲームステートをシミュレートし、グラウンドトゥルースとして権威あるゲームステートを確定させます。サーバーはこの権威あるステートをクライアントに返します。

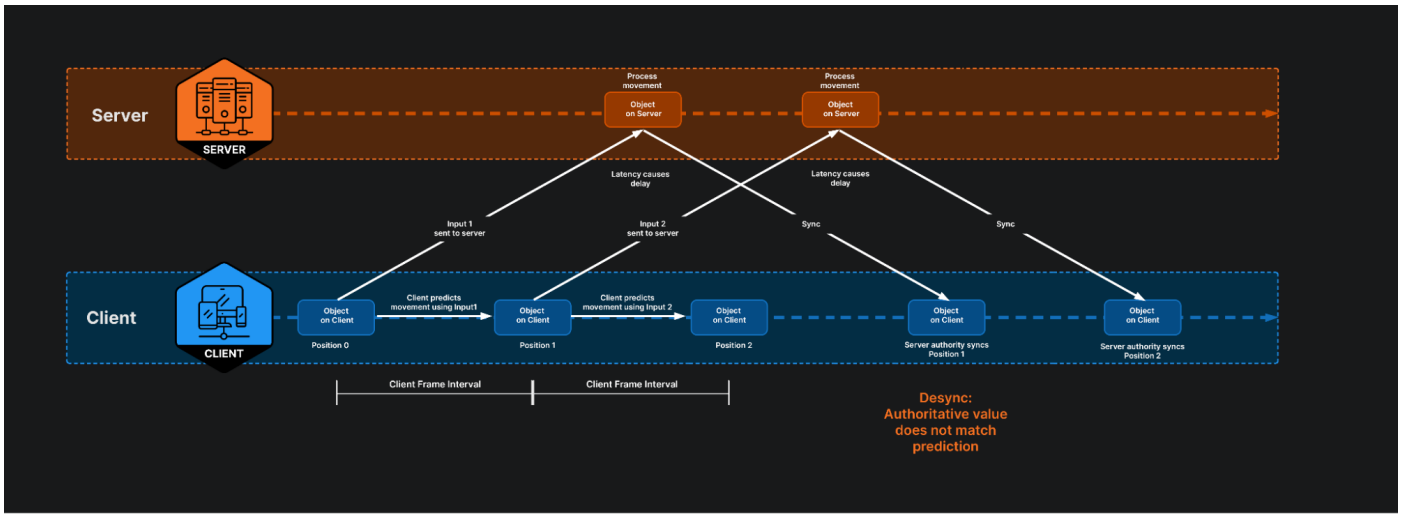


サーバーは権威あるステートを返します。

リコンシリエーションとロールバック

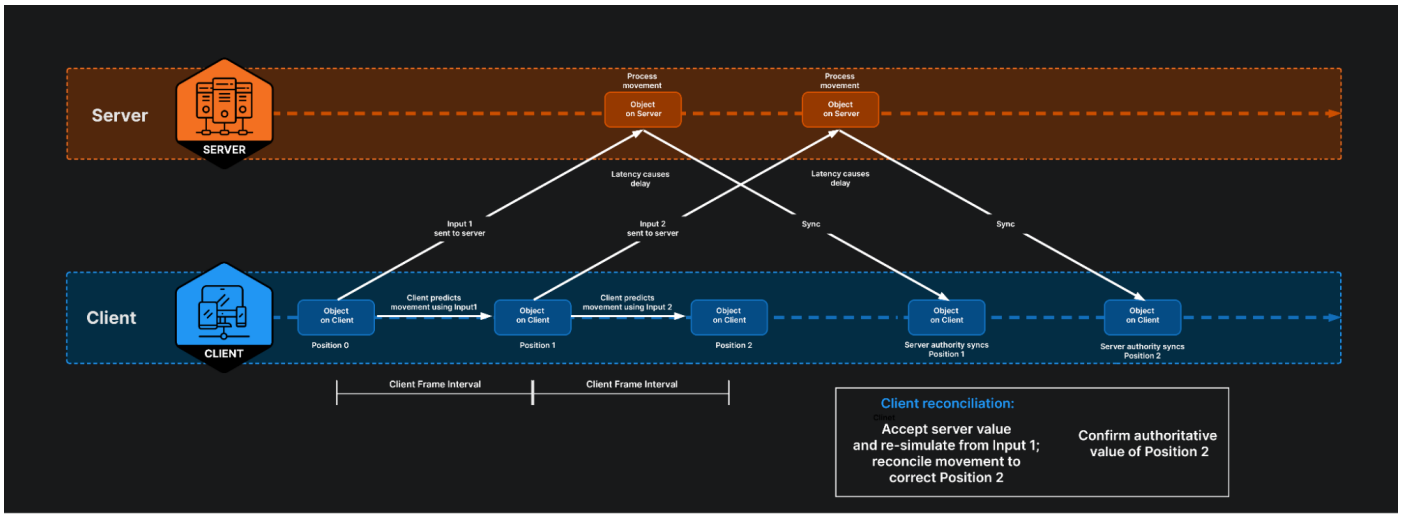
次に、クライアントは権威あるステートと先読みしたステートを比較し、違いを探します。2 つのステートが十分に近い場合は何も起こらず、クライアントは再生を続行します。

重大な不一致 (" 非同期 " とも呼ばれる) がある場合、クライアントは、サーバーの権威あるステートに合わせてステートを修正する方法を決定する必要があります。この修正プロセスを**リコンシリエーション**と呼びます。



クライアントは不一致または " 非同期 " のステートを受信しています。

待ち時間をハンドルし、それを補うために、クライアントは一定数のフレーム分の入力履歴と予測ステートを保存しています。そのため、非同期が発生した際に、クライアントはそのステートをサーバーから最後に認識された正しいステートにロールバックできます。その後、正しい入力を使用して、その時点からゲームを再シミュレートします。これにより、クライアントのステートがサーバーと再び同期されます。



クライアントが非同期のリコンシリエーションを行います。

どのフレームでも、クライアントは複数フレーム分を再度シミュレートしなければならない場合があります。そのため、ネットワーク接続されたアプリケーションの計算コストが高くなることがあります。それでも、このリコンシリエーションとロールバックは、ネットワーク待ち時間があってもプレイヤーにとって滑らかで一貫した体験を維持するのに役立ちます。

Netcode for GameObjects におけるクライアントサイド先読み

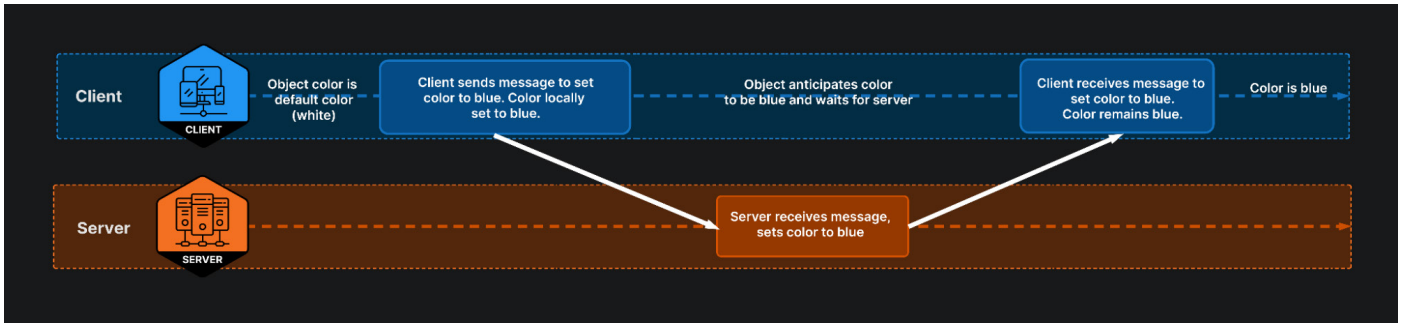
Netcode for GameObjects は、クライアント先読みをサポートしています。これは、完全なクライアントサイド予測やリコンシリエーションを行わずに、待ち時間を処理する簡易的なモデルです。このモデルは以下のコンポーネントを使用します。

— `AnticipatedNetworkVariable<T>` は、整数、浮動小数点、色などのスカラーデータ型や単純データ型に使用されるジェネリックコンポーネントです。体力、スコア、アイテムステートなど、Transform 以外のプロパティに適しています。

— `AnticipatedNetworkTransform` は、`AnticipatedNetworkVariable` と同様に機能しますが、位置、回転、スケールなどの Transform データ用に設計されています。

クライアント先読みにより、クライアントはサーバーからの権威ある更新を待っている間に、ユーザーに視覚的なフィードバックを即座に提供できます。これにより、ゲームの応答性が向上します。このシンプルな ColorTrigger の例では、プレイヤーがオブジェクトの色を白から青に変更すると、サーバーが変更を確認するのを待つ間に、クライアントは即座にその色を視覚的に更新できます。

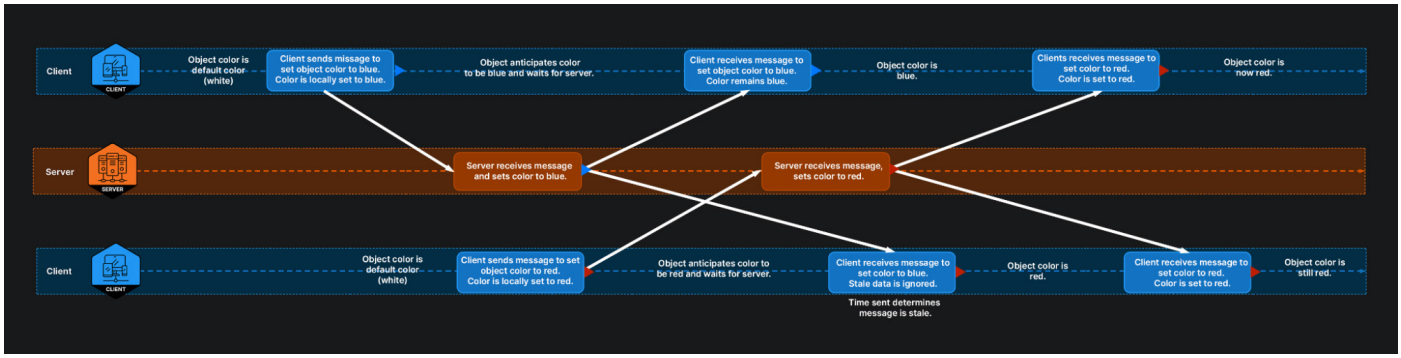
クライアントとサーバー間での先読みは、以下のように行われます。



クライアントはゲームステートを先読みします。

`AnticipatedNetworkVariable<T>` と `AnticipatedNetworkTransform` は、どちらも値を先読み (ビジュアル) ステートと権威あるステートに分けることで機能します。

先読みとは、クライアントが予測するステートを指し、ローカルでプレイヤーに即座に視覚的なフィードバックを提供します。権威あるステートはサーバーによって決定されます。サーバーの更新が届くと、クライアントは先読みステートと権威あるステートを比較します。これらが大幅に異なる場合、クライアントはサーバーに合わせてステートを調整し、すべてのクライアントにわたって一貫性を確保します。



クライアント先読みでは、古いデータを無視できます。

クライアント先読みは、" 古いデータ " にも対応する必要があります。この古いデータとは、クライアントが最後にリクエストを送信する前に、サーバーで発生したアクションが反映された更新情報のことです。Netcode for GameObjects では、`StaleDataHandling` プロパティを使用して以下 2 つの方法でハンドルできます。

— `StaleDataHandling.Ignore` は、古いデータを無視し、先読みされた値を保持します。これは、ステートが急激に変化し、ビジュアルのちらつきが発生している場合に便利です。

— `StaleDataHandling.Reanticipate` は、古いデータも他のサーバー更新と同様に扱い、ロールバックと再先読みをトリガーして、一貫性を維持するためにプレイヤー入力を再生します。

サーバーの値が予測値と異なる場合にビジュアルのちらつきを防ぐには、両方のコンポーネントで利用可能な Smooth メソッドを使用します。

Smooth には、スムージングプロセスの開始値、最終値、継続時間が必要です。これにより、ネットワーク待ち時間が発生しても、滑らかで応答性の高いビジュアル体験を維持できます。

クライアント先読みはゲームプレイの応答性を向上させますが、この簡易的なアプローチではあらゆるケースの待ち時間やネットワーク問題に対応できるとは限りません。

真のロールバックとリコンシリエーションには、同じ入力が常に同じ結果をもたらす決定論的な物理演算システムが必要であることに注意してください (以下参照)。これは、ゲームステートを正確にロールバックおよび再シミュレートするために不可欠です。決定論性がないと、サーバーとクライアントのシミュレーションに不一致が生じる可能性があります。

Netcode for GameObjects は、真のクライアント予測とラグ補償に必要な決定論的物理演算をサポートしていません。代わりに、Netcode for GameObjects はクライアント先読みとスムージングに重点を置いた、よりシンプルなソリューションを提供します。これにより、完全なロールバックシステムがなくても応答性を高めることができます。

決定論的な物理演算

決定論的な物理演算システムは、同じ初期条件と入力があれば、物理演算シミュレーションが常に同じ結果をもたらすことを保証します。

ただし、Netcode for GameObjects は、Unity のビルトイン物理演算エンジンを使用しており、これは決定論的ではありません。そのため、同じ入力でシミュレーションを再実行しても、物理演算システムは同一の結果を保証しません。

一方で、Netcode for Entities は決定論的なシミュレーション機能を提供する Unity Physics と Havok Physics をサポートしています。これにより、Netcode for Entities で真のクライアントサイド予測が可能になります。

Netcode for Entities におけるクライアントサイド予測

Netcode for Entities は、クライアントサイド予測とラグ補償をハンドルする上級者向けのツールを提供しています。各エンティティに対して、クライアントとサーバーの両方で同じシミュレーションコードが実行されます。これにより、クライアントはプレイヤー入力に基づいてゲームステートを予測でき、サーバーの応答を待たずに即座にフィードバックを得ることができます。

クライアントはサーバーから最新のスナップショットを受け取ると、そのデータを使用して予測されたすべてのエンティティを更新します。このスナップショットを適用すると、クライアントは PredictedSimulationSystemGroup というシミュレーションを実行します。このシミュレーションでは、保存された最も古いティックから現在のターゲット時間までを処理し、ゲームステートをロールバックおよび再シミュレートして、クライアントがゲームステートを正確にシミュレートできるようにします。このロールバックと再シミュレーションのプロセスは、クライアントが不一致を修正し、サーバーの権威あるステートとの同期を維持するのに役立ちます。

サーバー側では、予測ループが 1 フレームごとに 1 回実行されます。サーバーは権威あるゲームステートを更新しその、更新されたステートをクライアントに返します。

クライアントは、入力と予測されたステートの履歴を保存します。非同期が発生すると、クライアントはサーバーから受け取った最後の正しいステートにロールバックし、正しい入力を使用してゲームを再シミュレートします。これにより、クライアントとサーバーの " グラウンドトゥルース " の同期が維持されます。

Netcode for Entities は、以下のような上級者向けの物理演算機能もサポートしています。

- 複数の物理演算ワールド : これは、ネットワーク全体に複製する必要のないローカル限定の物理演算シミュレーションを可能にします。
- カスタム物理演算プロキシ : これを使用すると、クライアントにのみ存在する物理演算オブジェクト (例 : デブリ) とゴーストが相互作用できるようになります。
- 決定論的な物理演算シミュレーション : これは、クライアントとサーバーのシミュレーションが同じ入力に基づいて同じ結果を生成することを保証し、クライアントのステートとサーバーのステート間で一貫性を維持します。

GhostPredictionSmoothingSystem は予測エラーを平滑化するために使用され、予測されたステートと権威あるステートの間を遷移させることができます。また、カスタムスムージングを行うオプションもあります。

予測、ロールバック、ラグ補償、スムージング手法を組み合わせることで、待ち時間が与える影響を最小限に抑え、ゲームの整合性と応答性を維持するのに役立ちます。

Netcode for GameObjects と Netcode for Entities によるクライアント予測のハンドル方法を比較します。

機能	Netcode for GameObjects	Netcode for Entities
予測方法	クライアント先読み : クライアントは , サーバーの応答を先読みします。	完全なクライアント予測 : クライアントは、サーバーと同じシミュレーションコードを実行します。
待ち時間の軽減	サーバーの結果を先読みし、遷移を滑らかにする	ロールバックと再シミュレーションを使用して、非同期を修正する
スナップショット	明示的には使用されない	スナップショットを使用して、特定の時点におけるゲームステート表す
ラグ補償	StaleDataHandling オプションを使用した簡易的なモデル	衝突ワールドを参照した包括的なラグ補正
物理演算インタラクション	先読みされた Transform を使用した、クライアンサイド予測に限定	予測された物理演算ワールドとクライアントに限定されるインタラクションをサポート
Smoothing	先読みされた値に Smooth メソッドを使用	`GhostPredictionSmoothingSystem` を使用して、予測ステートと権威あるステートの間の遷移を滑らかにする
用途	即時の視覚的フィードバックを必要とするシンプルなゲームに最適	正確なステート同期を必要とする複雑なゲームに最適

Netcode for GameObjects と Netcode for Entities にはどちらも、クライアント予測をハンドルし、マルチプレイヤーネットワーキングで起きる待ち時間の問題を軽減するためのメカニズムが備わっています。Netcode for GameObjects は、クライアント先読みを使用したシンプルなモデルを提供する一方で、Netcode for Entities は、完全な予測、ロールバック、ラグ補償を備えたより上級者向けのシステムを提供します。

Netcode for Entities の用語

Netcode for Entities は、Entity Component System (ECS) と Data-Oriented Technology Stack (DOTS) を使用します。これらは Netcode for GameObjects で採用されている MonoBehaviour ワークフローとは異なります。見慣れない可能性のある用語をいくつか紹介します。

エンティティ : エンティティは ECS における基本のデータ単位で、ゲーム内の個別のゲームオブジェクトやコンポーネントを表しますエンティティは軽量で、動作を持たず、データのみが含まれます。

ゲームワールド : ゲームワールドは、ゲームが展開される全体の環境を指します。これには、すべてのエンティティ、そのステート、エンティティのインタラクションを管理するルールが含まれます。

衝突ワールド : 衝突ワールドは、ゲームワールド内のすべての物理オブジェクトのステートであり、その位置、速度、インタラクションなどが含まれます。これは衝突判定と物理演算シミュレーションに使用されます。

スナップショット : スナップショット (" ゴーストスナップショット " とも呼ばれる) は、特定の時点におけるゲームステートを表すデーター式のことです。スナップショットを通じてゲームステートを更新することで、クライアントとサーバーを定期的な同期を維持します。

ゴースト : ゴーストは、クライアントとサーバー間で複製されるネットワーク接続されたエンティティです。毎フレーム、サーバーは全ゴーストの最新ステートのスナップショットをクライアントに送信します。ゴーストは、サーバーとクライアントの間でエンティティのステートを同期するために使用され、すべてのプレイヤーが一貫したゲームワールドビューを持てるようにします。

予測ゴースト : 予測ゴーストは、ローカルでシミュレートされたクライアント側のエンティティであり、プレイヤーアクションに対して視覚的なフィードバックを瞬時に提供し、待ち時間の違和感を軽減します。プレイヤーが直接制御するエンティティや、即時のフィードバックが必要なその他の相互作用するエンティティに使用します。

補間ゴースト : 補間ゴーストは、サーバー側のエンティティをクライアント上に表します。クライアントは、サーバーから受け取ったスナップショットを基にステートを表示し、それらをブレンドして待ち時間によるジッターを最小限に抑えます。他のプレイヤーキャラクター、NPC、サーバーで制御されているエンティティなど、プレイヤーが直接制御していないエンティティで、即時のフィードバックを必要としない場合に使用します。

ネットワークを介したゲームのテストとデバッグ

マルチプレイヤーゲームのテストとデバッグは、シングルプレイヤーゲームとは異なります。ネットコードプロジェクトにおける一般的なワークフローを把握しておきましょう。

— **ローカルテスト**では、複数のゲームインスタンスを実行し、プレイヤー間のインタラクションを評価します。プレイヤービルド、Multiplayer Play Mode パッケージ、Network Scene Visualization を使用して、アプリケーション開発を進めます。

— **ネットワーク状態のシミュレーション**には、スクリプトまたは NetcodeTransport コンポーネントを使用します。これにより、待ち時間、ジッター、パケットロスといった、実際のネットワーク問題を模倣できます。

— **クライアント接続管理**は、クライアントの参加、再接続、接続切断時の動作をハンドルします。

— **ログ出力**は、デバッグツールを使用して問題のトラブルシューティングを監視します。

— **コマンドラインヘルパー**は、特定のロールやネットワーク状態でゲームインスタンスを起動し、テストを自動化します。

ローカルテスト

マルチプレイヤーゲーム開発には、ローカルテストが不可欠です。テストでは、複数のゲームインスタンスをシミュレートし、異なるプレイヤーがネットワーク環境でどのように相互作用するかを模倣します。

プレイヤービルド

プレイヤービルドを使用すると、複数のゲーム実行ファイルインスタンスを実行して、ゲームをホストしたりゲームに参加したりできます。これを Unity エディターと一緒に実行することで、1 つのデバイスでゲームを同時にホストおよび参加させ、マルチプレイヤーシナリオをシミュレートできます。

Multiplayer Play Mode (MPPM)

Unity 6 に含まれている Multiplayer Play Mode (MPPM) パッケージを使用すると、1 つの開発デバイスで最大 4 人のプレイヤーをシミュレートできます。このとき、すべて同じソースアセットを使用しており、個別のプレイヤービルドが不要になるため、テスト時のビルド時間が短縮されます。

macOS ユーザー向け

macOS ユーザーの場合、アプリの複数のインスタンスを実行するには、コマンドラインでのコマンドが必要です。ターミナルで `open` コマンドを使用して、アプリケーションの別のインスタンスを起動します。

例えば、ターミナルで `open -n YourAppName.app` を実行すると、`YourAppName` アプリケーションの別のインスタンスが起動します。

ネットワーク状態をシミュレートする

ローカルでテストする際は、すべてのゲームインスタンスが同じネットワークインターフェースで実行されます。このとき、クライアント間の待ち時間はほとんど、またはまったく発生しないため、実際の環境をシミュレートするには、条件を人為的に導入する必要があります。

待ち時間、ジッター、パケットロスは、ゲームプレイに影響を及ぼします。ネットワーク環境が理想的でない状態でアプリケーションをテストすることで、最終的なビルドがインターネット上でも問題なく動作することを確認できます。

テストする状態は、対象プラットフォーム、地域、ゲームの設計などの要因によって決まります。まず、待ち時間の値をデスクトップでは 100 - 150 ミリ秒程度、モバイルでは 200 - 300 ミリ秒程度とし、パケットロスはどちらも 5 - 10% 程度とします。ジッターやパケットロスを含むテストを行うことは、現実的な不安定さをもたらすため不可欠です。詳細なガイドラインについては、こちらのドキュメントページを参照してください。

エディター内でローカルにテストする場合、Multiplayer Tools の Network Simulator ツールと Multiplayer Play Mode をともに使用します。

開発ビルドをテストする場合、Network Simulator ツールといくつかのカスタムコードを使用して、ビルド内で悪いネットワーク状態を再現することを推奨します (Network Simulator ウィンドウは、エディターでのみ機能します)。

リリースビルドをテストする場合、Windows 環境なら clumsy 、macOS または iOS 環境なら Network Link Conditioner の使用を推奨します。macOS では、Network Link Conditioner の代わりに dummynet を使用できます。これはスクリプト対応で優れた制御性を備えており、オペレーティングシステムにパッケージ化されています。

詳細については、システム全体のネットワークコンディショナーを参照してください。

クライアント接続をテストする

ネットワークを介したゲームでクライアント接続管理をテストすることは、バグを回避し、滑らかなゲーム体験を提供するために重要です。テストと注意が必要な点を以下に紹介します。

クライアントの接続時 :

— テストケースには、クライアントの新しいゲームセッションへの参加、離脱またはホスト後の再参加、進行中のゲームへ途中参加、接続拒否時の対応が含まれます。

— クライアントの過去のステート状態が接続に影響するかどうか、ゲームステートがサーバーから正しく複製されるかどうかを注意深く確認します。

— サーバーが再接続または途中参加を適切にハンドルできているかどうかを確認します。

クライアントの接続切断時 :

— クライアントの正常なシャットダウン、タイムアウト、ホスト / サーバーへの接続切断時の影響をテストします。

— ゲームセッションに結び付けられたオブジェクトが破壊されていない場合には正しくリセットされ、クライアントが新しいゲームに再接続できることを確認します。

ホスト / サーバーのセッション開始時 :

— 新しいゲームセッションの開始時、前のセッションをシャットダウンした後をテストします (特に、クライアントホスト型ゲームの場合)。

— 新しいセッションを開始する前のアプリケーションの状態が、ゲームに影響を与えるかどうかを注意深く確認します。

ホスト / サーバーのシャットダウン時 :

— ホスト / サーバーの正常なシャットダウンをテストします (特に Unity Game Services やロビーサービスなどの外部サービスを使用している場合)。

— クライアントにシャットダウンが通知され、外部サービスに正しく情報が通知されることを確認します。

これらのテストケースを注意深く確認することで、安定した快適なネットワークゲーム体験を維持できます。

詳細については、クライアント接続管理をテストするを参照してください。

マルチプレイヤーゲームにおけるデバッグ手法

マルチプレイヤーゲームをデバッグする際には、通常のゲーム開発における基本的な知識がすべて役に立ちます。ただし、マルチプレイヤーゲーム開発によく見られる特定のシナリオでは、特別なコツやアプローチが必要になります。

以下に、Unity でマルチプレイヤーゲームを開発する際に役立つ手法をリストで示します。

— **デバッグ描画手法 :**Debug.DrawRay や Debug.DrawLine を使用してデバッグ用のラインを描画し、ネットワーク上の位置、動きの意図、オブジェクトのインタラクションを示します。これは、マルチプレイヤーゲームプレイの横並びの録画と組み合わせて使用すると有効です。

— **Netcode 対応の Line Renderer:** 状況によっては、特定の方向や値など、プロジェクトに関連する便利なデバッグメトリクスを示す視覚的なフィードバックがあると便利です。Netcode 対応の Line Renderer の実装に関するスクリプト例を参照してください。

— **テキストベースのログ出力 :** テキストベースのログ出力では、不可視のイベント（RPC など）や情報を追跡できます。ログメッセージにネットワークティックやクライアント ID を含めることで、ログを読む際にタイムラインをビルドしやすくなります。

— **ネットワークコンディショニング :** アプリケーションレベルのネットワークワークコンディショニングには、Network Simulator ツールを使用します。これにより、人為的なネットワーク状態をシミュレートでき、待ち時間、ジッター、パケットロスに特化したエラーのテストに役立ちます。詳細については、システム全体のネットワークコンディショナーを参照してください。

— **画面録画 :** クライアントとサーバーの両方のインスタンスを同時に記録して、リアルタイムのゲームプレイを比較します。デバッグビルドでは、各フレームにクライアント ID と現在のフレーム番号をスタンプしておくと、並べて比較する際の視覚的な基準になります。

— **固定時間ステップの増加 :** 優れたデバッグレンダリングとログ出力を使用しても、フレームを 1 つずつ確認している場合でも、何が起きているのか理解するのが難しいことがあります。**FixedTimeStep** の設定値を大きめに（例：0.2）することで、各フレーム中のゲームの動作をより明確にできます。

— **ブレークポイントの使用 :** ブレークポイントを使用してゲームをデバッグしていると、このモードを長時間続けた場合に接続がタイムアウトすることがあります。これは、ゲームが一時停止するためであり、接続の切断を防ぐにはタイムアウトの値を一時的に増やします。

ヒントと手法の詳細については、マルチプレイヤーゲームのデバッグ手法とコツに関するガイドを参照してください。

コマンドラインヘルパー

Unity エディター内でマルチプレイヤービルドを繰り返し起動してテストすると、時間がかかることがあります。コマンドラインツールを使用して、エディター環境外でのマルチプレイヤーゲームのビルドを自動で起動およびテストすることを検討してください。

このサンプル NetworkCommandLine スクリプトは、コマンドラインツールの使用を開始する際に役立ちます。NetworkCommandLine コンポーネントをゲームオブジェクトにアタッチすると、このスクリプトがビルドに含まれ、コマンドラインからアクセスできるようになります。

Player Settings の **Settings for PC, Mac, & Linux Standalone** の下にある **Resolution and Presentation** を選択します。**Resolution** を **Windowed** に設定します。これにより、複数のゲームインスタンスを並べてテストしやすくなります。

NetworkCommandLine スクリプトでは、まずゲームがエディター外で実行されているかどうかをチェックします。実行中であれば、コマンドライン引数を読み取り、実行するモード (サーバー、ホスト、クライアント) を決定し、それに対応するサービスを開始します。これにより、コマンドラインから特定のネットワークロールでゲームビルドを起動できます。

File > Build Settings からバイナリをビルドしたらコマンドラインでテストします。

Windows の場合：

コマンドプロンプトを開き、ビルドを保存したディレクトリに移動させます。特定のコマンドを使用して、ゲームのサーバーまたはクライアントのインスタンスを起動し、必要に応じてパスを調整します。例：

```
<Path to Project>\HelloWorld.exe -mode server
<Path to Project>\HelloWorld.exe -mode client
```

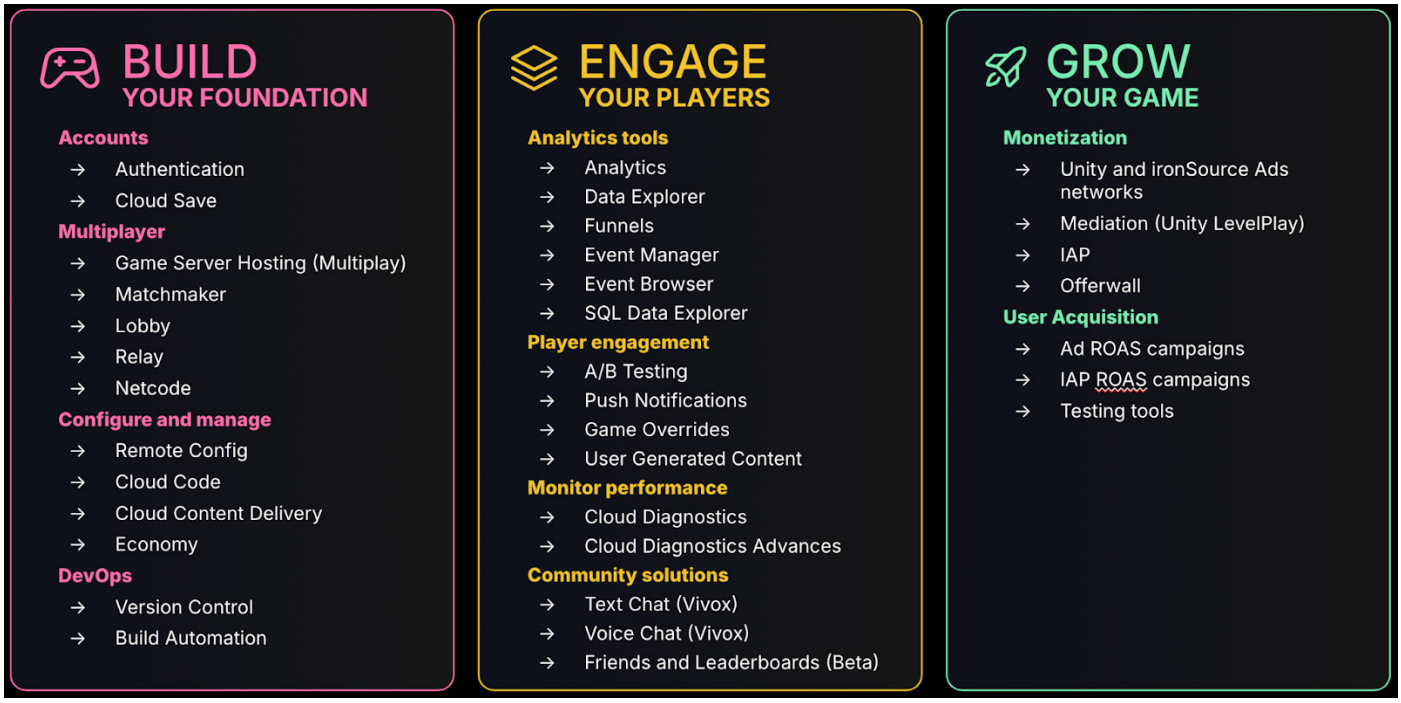
macOS の場合 :

macOS では、ターミナルを開き、上記と同様のコマンドを使用します。ただし、macOS のディレクトリ構造に合わせてパスを調整します。例：

```
<Path to Project>/HelloWorld.app/Contents/MacOS/<Project Name> -mode server
<Path to Project>/HelloWorld.app/Contents/MacOS/<Project Name> -mode client
```

任意ですが、**-logfile** フラグを付けると、ゲームインスタンスの出力をテキストファイルにログ出力でき、追跡や分析がしやすくなります。

マルチプレイヤーサービス



Unity では、ネットワークを介したマルチプレイヤーゲーム開発を容易にするためのサービスを多数提供しています。まずは Multiplayer Services パッケージのドキュメントを確認して、各サービスについて学び始めましょう。このパッケージを使用すると、複数のマルチプレイヤーサービス間の依存関係の管理が簡単になります。例えば、以下のようなことが可能になります。

- Unity Gaming Services をゲームに組み込むマルチプレイヤー要素をすばやく加えられます。Lobby、Relay、Distributed Authority、Matchmaker、Multiplay Hosting を設定できます。
- 新しいセッションシステムは、マルチプレイヤーゲームループ向けに、クラウド上にあるシンプルな共有バックエンドを提供し、プレイヤーをグループ化して、共有のセッション / プレイヤーステートを管理します。
- ピアツーピア (P2P)、Dedicated Game Server、Distributed Authority によるホスト型オンラインセッションを作成して管理できます。プレイヤーは、マッチメイキング、参加コード、またはアクティブなセッションのリストを参照することでセッションに参加できます。

各サービスについて簡単に見ていきましょう。

マッチメーカー

マッチメイキング とは、バランスのとれた快適なゲーム体験を実現するために、スキルレベルや地理的位置などの特定の基準に基づいて、プレイヤーを他のプレイヤーやゲームセッションに接続するプロセスです。

Unity の Matchmaker サービスを実装するには、いくつかのステップがあります。

- **マッチメイキングの基準を定義する :** プレイヤーのマッチに使用するパラメーターを決定します。これには、スキルレベル、地理的近接性、待ち時間、プレイヤーの好みなどが含まれます。
- **プレイヤープロファイルと評価 :** 各プレイヤーのプロファイルを保持し、そこにマッチメイキングの基準を加えます。評価システム (例 : イロレーティング) を使用してプレイヤーのスキルレベルを評価し、バランスの取れたマッチを実現します。
- **マッチメイキングのリクエスト :** プレイヤーがマッチをリクエストすると、マッチメイキングサービスは定義された基準に基づき、適切な対戦相手やチームメイトを検索します。このとき、対応可能なプレイヤーのプールにクエリを実行し、可能な限り最適なマッチを見つけます。
- **マッチの割り当て :** 適切なマッチが見つかったら、プレイヤーはゲームセッションに割り当てられます。このとき、すべてのプレイヤーが接続され、ゲームを開始できる状態になっていることを確認します。
- **動的調整 :** 特定の基準を満たしていない場合、マッチメイキングサービスはパラメーターを動的に調整して、品質を犠牲にすることなく迅速にマッチを成立させます。

これらの要素を考慮しながら、Unity の Matchmaker を使用して、公平かつ競争性の高いマッチを作成しましょう。

Lobby

ロビーは、プレイヤーが集まり、ゲーム設定を構成し、ゲーム開始の準備を行うためのゲーム開始前エリアです。Lobby サービスは、マルチプレイヤーゲームのさまざまなシナリオにおいて、プレイヤー同士が検索してつながることができる手段を提供します。よくある例をいくつか紹介します。

- 利用可能なゲームセッションのリストを参照して、セッションを選択して参加する
- フレンドと参加コードを共有して、ゲームセッションに接続できるようにする
- Quick Join を使用して、利用可能なマッチを検索して参加する
- 公開ロビーまたは非公開ロビーを作成し、ゲーム内のフレンドリストのプレイヤーに招待を送信する
- ゲームサーバーからロビーをホストして、サーバーセッションへのアクセスを管理および制限する
- 特定の要件 (例 : ゲームモード、マップタイプ) に一致するロビーにクエリを実行する

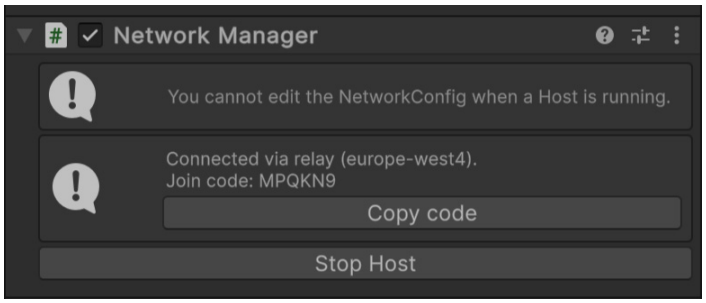
Game Lobby サンプルでは、Lobby パッケージと Relay パッケージを使用して、Vivox ボイスチャットなどの典型的なゲームロビー体験を作成する方法を紹介しています。プレイヤーはロビーをホストでき、そのロビーには、公開ロビーリストまたはロビーコードを使用して他のプレイヤーが参加できます。その後、Relay を介して接続し、Unity Transport を使用した基本的なリアルタイム通信が可能になります。

Relay

Unity Relay は、参加コードシステムを通じて複数のプレイヤーを接続するプロセスをシンプルにします。ホストがゲームセッションを設定すると、一意の参加コードが生成され、これをフレンドやチームメイトと共有することで、そのコードを使用してセッションに接続できます。このシステムは、IP アドレスなどの機密情報を非公開にすることでプライバシーを確保しながら、接続プロセスを効率化します。

ピアツーピアのマルチプレイヤーゲームでは、NAT (ネットワークアドレス変換) やファイアウォールといったネットワークの問題により、クライアントの接続が困難になる場合があります。直接接続では、複雑なネットワーク設定を必要になることが多く、プレイヤーの IP アドレスが公開されるため、セキュリティとプライバシーの問題が生じます。

Relay は中継サーバーとして動作し、クライアント間の接続を安全かつシンプルにすることで、これらの問題を解決します。専用サーバーの必要性がなくなり、開発オーバーヘッドが削減されます。Relay を使用すれば、参加コードを生成するだけで、プレイヤーがすぐに接続できます。



接続後、Relay は参加コードを提供します。

詳細については、Unity Relay のスタートガイドを参照してください。

Multiplay ホスティング

Multiplay Hosting は、インフラストラクチャの大規模な実行および運用に伴う複雑さを排除するため、開発チームは魅力的なプレイヤー体験の作成に集中できます。また、以下のようなことを実行できます。

- サーバーの健全性やその他の分析データを追跡します。
- ダウンタイムなしのパッチ適用でサーバーを更新します。
- サービス品質 (QoS) データに基づいて、最適な体験を提供できるサーバーにプレイヤーを配置します。
- Docker と Multiplay Hosting コンテナレジストリを使用して、ビルドをコンテナ化します。

Vivox

Vivox を使用すると、プレイヤーがゲーム内のボイスチャットまたはテキストチャットを通じて会話することができ、協力型や対戦型のマルチプレイヤー体験を向上させます。Vivox では、マネージホスト型ソリューション通じてテキストチャットやボイスチャットをサービスに統合できます。音声認識やチャットフィルタリングなどのアクセシビリティ機能と規制対応機能も備えています。

Vivox は、ゲームが Unreal エンジン、Unity エンジン、カスタムエンジンでビルドされているかどうかに関係なく、複数のプラットフォームにわたるプレイヤー通信を可能にします。

ゲームに Vivox を組み込み、プロジェクト設定を行い、Unity Dashboard からプロジェクトにコミュニケーション機能を加えます。2D および 3D チャンネルで接続できるユーザー数に制限はありません。ユーザーは音声の音量制御、ミュートアクションの実行、チャンネルの管理などを行えます・

Megacity Metro のデモに、多数のマルチプレイヤーサービスがどのように統合されたかを説明する GDC セッション (YouTube 自動翻訳字幕推奨) を参照してください。

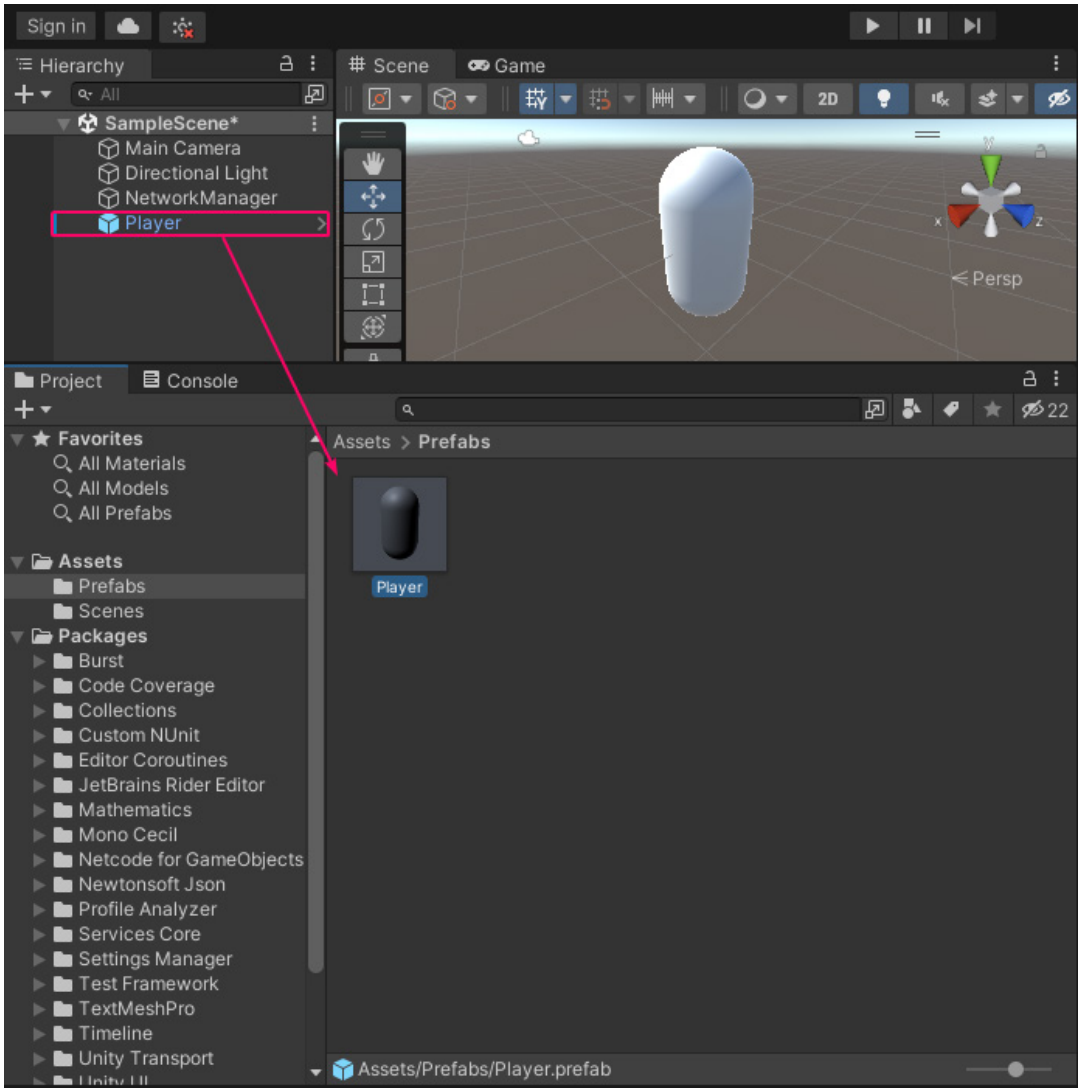
サンプルプロジェクトとリソース

Unity では、Netcode for GameObjects および Netcode for Entities の使用を開始するのに役立つ、さまざまなサンプルプロジェクトを用意しています。これらのリソースは、独自のアプリケーションにマルチプレイヤーネットワークを実装する方法のガイドとなる実践的な例を提供します。

Netcode for GameObjects 向けリソース

Unity Learn：Netcode for GameObjects の使用を開始する

この Unity Learn チュートリアルでは、Unity での基本的なマルチプレイヤーゲームのビルドおよびテスト方法のほか、リモートプロシージャコール (RPC) と NetworkVariable の使い方とテスト方法を実践的なステップで紹介しています。また、各モード (ホスト、クライアント、サーバー) 用のシンプルなユーザーインターフェースを作成する方法や、各モードの基本動作を加えたり制御したりする方法も学習できます。



Netcode for GameObjects に関する Unity Learn チュートリアルのスクリーンショット

Bitesize Samples

『Bitesize Sampvles』のリポジトリでは、ゲームで使用できるモジュール形式の一連のサンプルコードを提供しており、Netcode for GameObjects について理解を深めることができます。リポジトリには以下のようなものが含まれます。

— 2D スペースシューターサンプル :Netcode の NetworkVariabve やオブジェクトプールを使用した、物理演算による動きとステータス効果を学習できます。

『Bitesize Samples』の 2D スペースシューター

— Distributed Authority ソーシャルハブ : ゲームステートの制御と管理を複数のクライアントに分散させ、サーバーのロード軽減を可能にするトポロジーの設定方法を学習できます。

— マルチプレイヤーユースケースの概要 : このサンプルは、マルチプレイヤー環境での一般的なアクションを実行する方法を紹介しており、これを念頭に置きながらゲームに機能をビルドできます。

— クライアント駆動サンプル : クライアント主導の動き、ネットワークを介した物理演算、スポーンされるオブジェクトと静的に配置されたオブジェクトの比較、オブジェクトの親子関係の変更について学習できます。

— Addressables による動的なネットワークプレハブ : ランタイム時に新しいスポーン可能なプレハブを加えることが可能な、動的プレハブシステムについて学習できます。

Boss Room

『Boss Room』は、Netcode for GameObjects を使用してビルドされた 3D カジュアル協力型ゲームのサンプルプロジェクトで、マルチプレ v イヤーゲームのフローの背景にある概念やパターンを探るのに役立ちます。

『Boss Room』は、フル機能の協力型マルチプレイヤーゲームの一部を切り取ったサンプルです。

『Boss Room』は、実際に機能するゲームサンプルであると同時に、ネットワークを介したマルチプレイヤーゲームに関心のある開発者向けの学習ツールでもあります。『Boss Room』は、Netcode for GameObjects のワークフローを使用してビルドされた、Unity で最も長く稼働している本番対応のマルチプレイヤーサンプルです。このサンプルには本番レベルのコードが含まれており、Lobby や Relay といったホスティングサービスと統合されています。

『Boss Room』では、キャラクターの能力や特殊攻撃、ネットワークを介した物理演算、ステート追跡 (例 : 破損しやすいオブジェクト、スイッチ、ドア) など、ネットワークプレイにおけるゲームメカニクスを紹介しています。これらの手法は、理想的なネットワーク状態にない状況下でも、待ち時間を目立たせずに滑らかなゲームプレイを実現する方法を紹介しています。詳しく見てみましょう。

ゲームフローとステート管理 :『Boss Room』には、ロビーとゲーム内プレイの両方に対応するゲームステートの実用的な実装が含まれています。プレイヤー接続のハンドル、ネットワークプレイ用のシーンのロードとアンロード、正常な接続切断の管理方法を示しています。

UGS との統合 :『Boss Room』には、Relay、Lobby、Authentication など、いくつかの UGS 機能が統合されています。

ユーティリティスクリプトとツール : リポジトリには、クライアント権限、シーン管理ユーティリティ、ネットワークを介したオブジェクトプールの例など、他のプロジェクトで使用するために調整できる、さまざまなユーティリティスクリプトやツールが含まれています。

クライアンサーバーモデル：『Boss Room』は、プレイヤーの 1 人がホスト (サーバー) で、他のプレイヤーがクライアントというクライアントサーバーモデルを採用しています。ゲームロジックを一元化することでチートのリスクを軽減し、すべてのクライアントでゲームステートの一貫性を確保します。

『Boss Room』サンプルプロジェクトの開始メニュー

プロジェクトの詳細については、こちらのブログ記事と 4 部構成の YouTube ウェビナー(YouTube 自動翻訳字幕推奨)「Netcode for GameObjects でマルチプレイヤーゲームをビルドする」を参照してください。

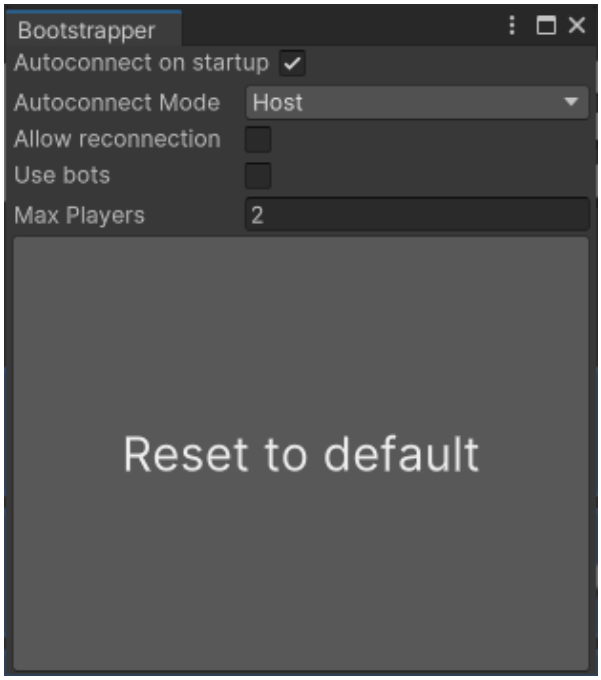
このシリーズでは、ネットワークを介したゲームプレイやオブジェクトのスポーン処理の実装方法など、基本的なゲームメカニクスとサーバー権限について説明します。また、プレイヤーアクションに対するゲームのレジリエンスを高め、帯域幅管理を最適化する方法も紹介しています。このリソースは、フル機能のマルチプレイヤープロジェクトを使用した実践的なハンズオンガイダンスを求めている開発者に最適です。

小規模な対戦型マルチプレイヤーテンプレート

Unity Hub から入手できるこのテンプレートは、Netcode for GameObjects と UGS を使用してマルチプレイヤープロジェクトを作成および公開するための出発点となります。

テンプレートには、各種ネットワークモード (ホスト、クライアント、サーバー) や動的な設定を使用してテストするのに役立つ Bootstrapper ツールが含まれています。

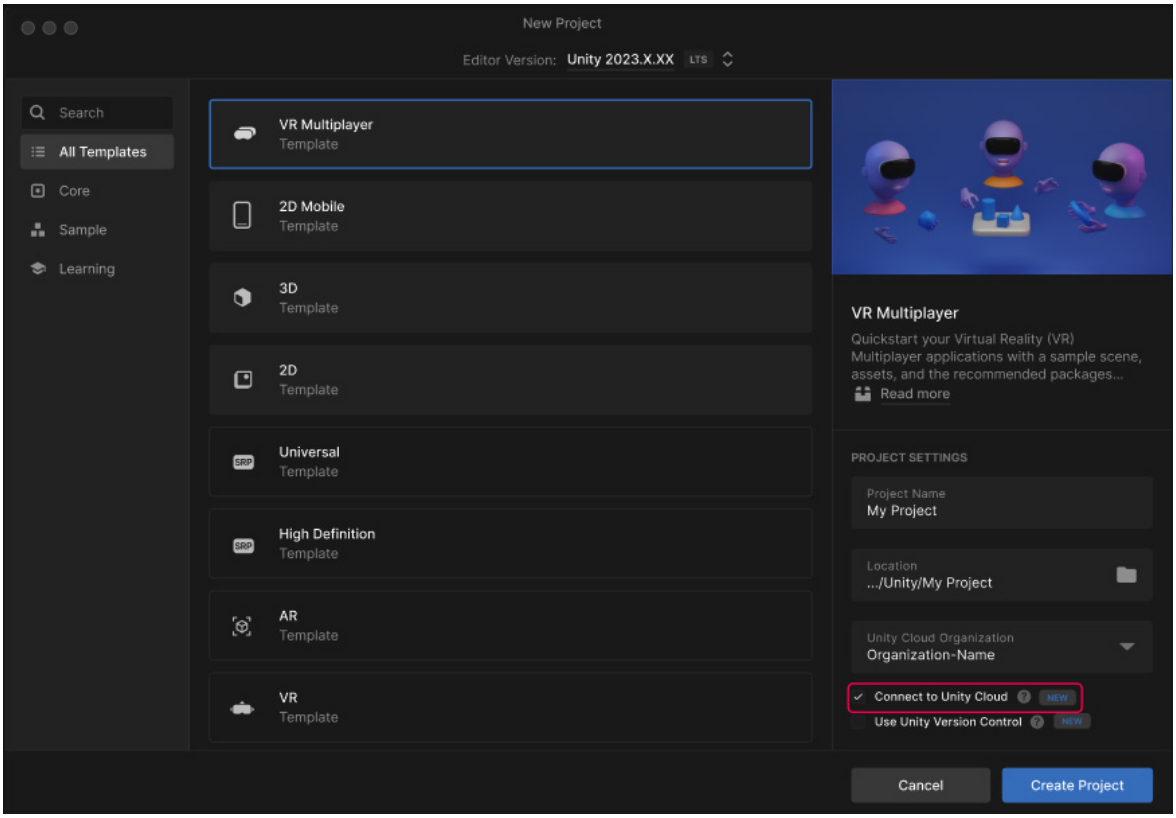
使用を開始するには、プロジェクトを開き、表示されるウェルカムダイアログの指示に従います。エディター内のチュートリアルに従うか、自由に検索して、**Multiplayer > Bootstrapper** メニュー項目から開始シーンをロードします。テストする内容に応じて Bootstrapper の各フィールドの値を調整し、再生モードを開始します。



Bootstrapper オプションを設定します。

チュートリアルや便利なリソースには、**Tutorials > Show tutorials** メニューからいつでもアクセスできます。

VR Multiplayer テンプレート



Unity Hub から VR Multiplayer テンプレートを開きます。

VR Multiplayer テンプレートは、OpenXR デバイスを対象とした新しいプロジェクトを開始するクリエイター向けに設計されています。ネットワークを介したインタラクション、ボイスチャット、ロビーなどに必要なものがすべて揃っています。Unity Gaming Services、Netcode for GameObjects、XR Interaction Toolkit を活用しており、Meta Quest プラットフォームや他の OpenXR 準拠デバイスでも機能するようビルドされています。VR Multiplayer テンプレートの主な機能は以下のとおりです。

- XR Interaction Toolkit と Netcode for GameObjects によるネットワークを介したインタラクション
- Vivox による音声コミュニケーション
- Relay と Lobby を使用して、誰とでも、どこからでもつながる
- ネットワーク対応のアバターと手
- OpenXR を介してクロスプラットフォームで動作する

Unity Hub から Unity 2022 LTS および Unity 6 Preview 用の Unity VR Multiplayer Template をダウンロードし、クイックスタートガイドを確認してください。

Netcode for Entities のリソース

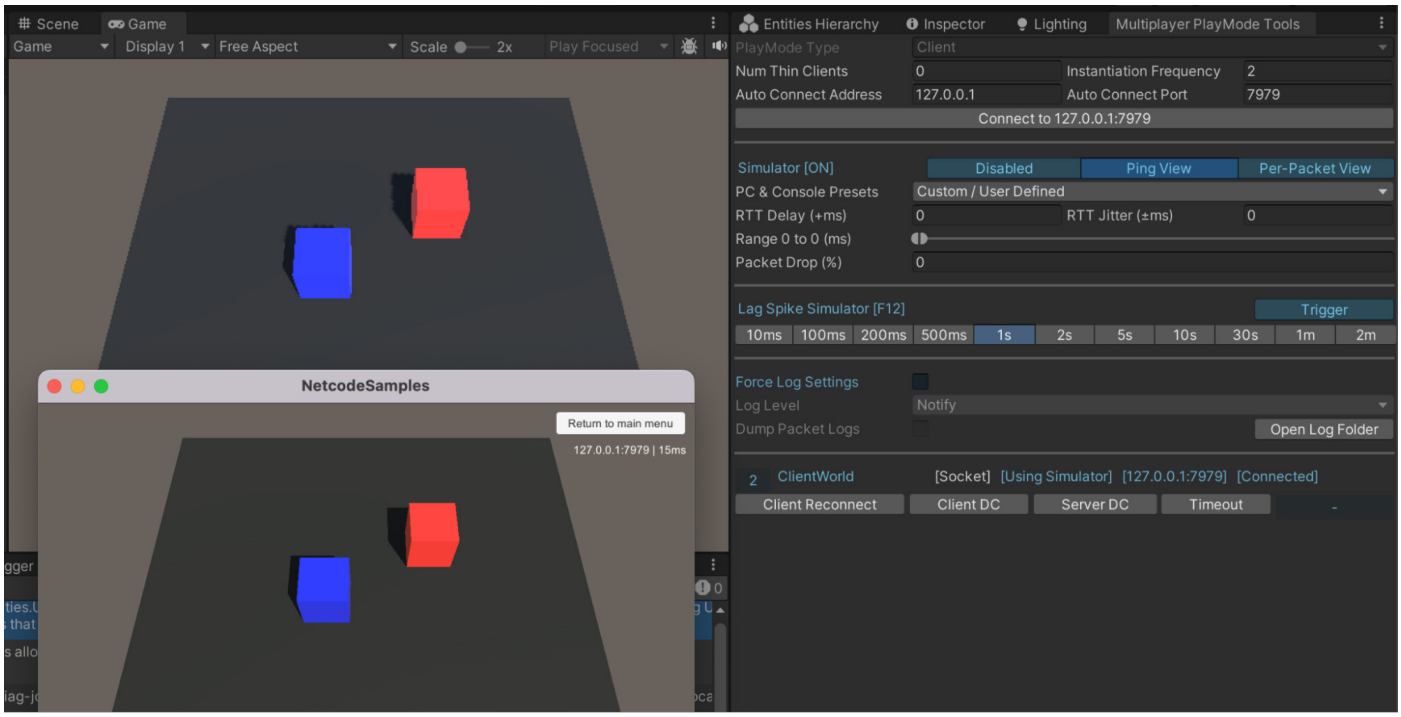
Netcode for Entities の使用を開始する

このオンデマンドウェビナー(YouTube 自動翻訳字幕推奨) では、Unity の大規模なクロスプラットフォーム対応のマルチプレイヤーゲームサンプルである『Megacity Metro』について掘り下げます。このサンプルは、Netcode for Entities と Unity Gaming Services を使用して作成されています。このウェビナーは、DOTS にすでに精通しており、大がかりなマルチプレイヤーゲームの制作を開始しようとしているユーザーを対象としています。

ECS Netcode サンプル

これらのサンプルでは、同期、接続フロー、Unity Physics との統合など、多くの基本的な機能および上級者向けの機能を紹介しています。まずは Networked Cube チュートリアルで以下を学びましょう。

- サーバーとの接続の確立。
- サーバーとの通信。
- サーバーでの同期エンティティのスポーン。
- サーバーとクライアントのスタンドアロンビルドの作成。
- エディター内での再生モードでのサーバーとクライアントの実行。



エディター内、そしてスタンドアロンビルドとして起動中の『Networked Cube』チュートリアル

ECS Network Racing

このマルチプレイヤーレーシングゲームのサンプルでは、Unity Netcode for Entities を使用するためのベストプラクティスを紹介しています。このデモでは、クライアントサイド予測、補間、ラグ補償を含む、クライアントサーバー型アーキテクチャが実装されています。

『ECS Racing』は、上級者向けのマルチプレイヤー機能を紹介しています。

Megacity Metro

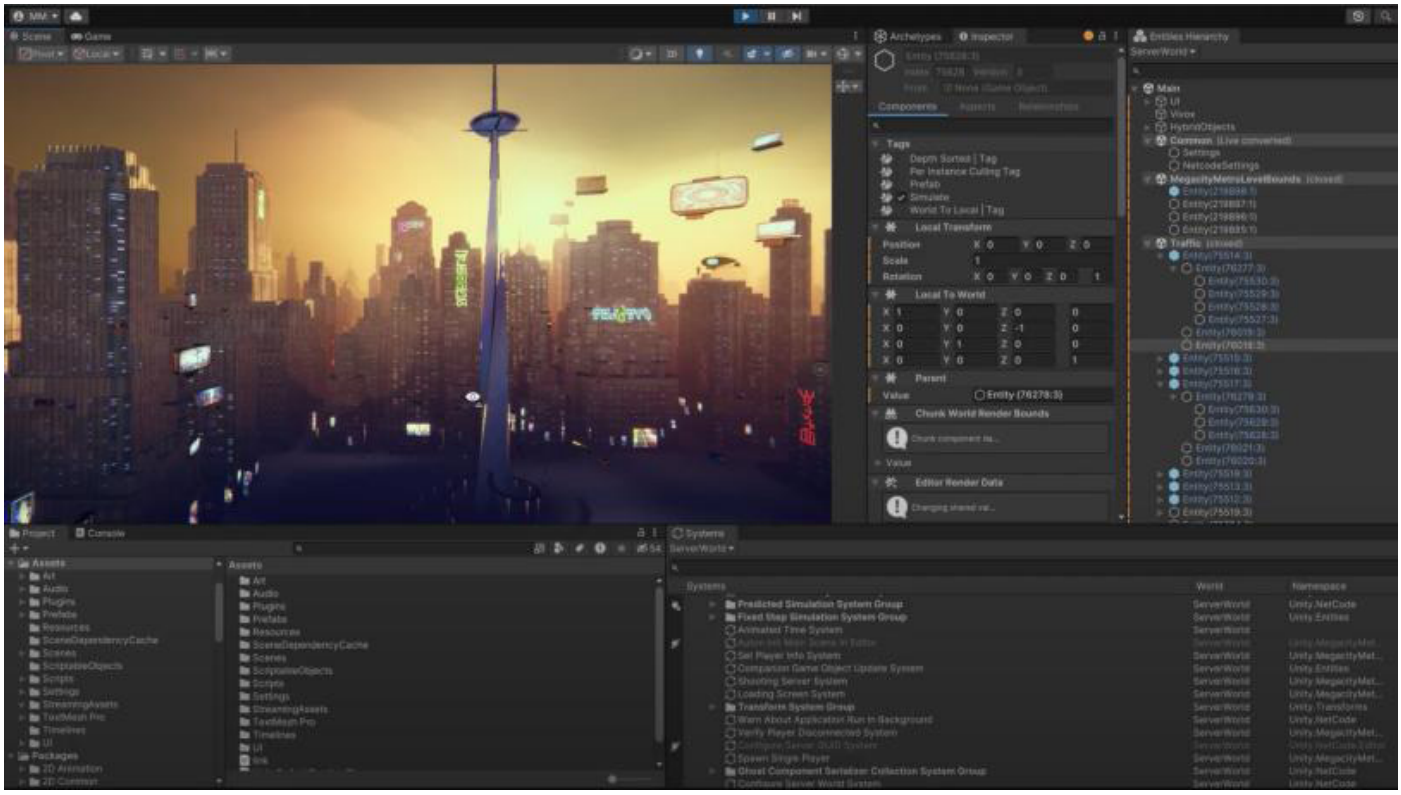
『Megacity Metro』(Unity 6、Unity 2022 LTS) は、Unity の最新テクノロジーを活用したスケーラブルな高同時接続対応クロスプラットフォームのデモで、Netcode for Entities パッケージが使用されています。この三人称視点のマルチプレイヤーアクションデモは、128 人以上の同時プレイに対応しています。

マルチプレイヤーサンプルは、サーバー権威型のゲームプレイ実装や、エンドツーエンドのマルチプレイヤーゲームでの UGS 活用をマスターしたい方に最適です。

Unity のデモ『Megacity Metro』では、128 人以上の同時プレイに対応しています。

ECS for Unity と マルチプレイヤーソリューションを使用した大がかりなゲームの開発に関心がある方は、こちらのデモを活用してください。ECS for Unity は、さらなる制御と決定論を必要とするより大がかりなゲームの実現を目指す、経験豊富な Unity クリエイターにとって価値あるソリューションです。

『Megacity Metro』は、補間、クライアントサイド予測、ラグ補償などの上級者向けのメカニクスを備えています。デモをダウンロードして、Multiplay Hosting、Matchmaker、Vivox Voice Chat などのサービスを試してみましょう。



『Megacity Metro』では、マルチプレイヤーアクションデモで Netcode for Entities を使用しています。

実験的な Multiplayer Services パッケージ

マルチプレイヤーゲームを構築するには、複数の製品やサービスを統合する必要があります。新しい **Multiplayer Services** パッケージ (**com.unity.services.multiplayer**) では、マルチプレイヤーサービス間の統合と依存関係の管理をシンプルにしつつ、製品と相互作用するための新しい方法を提供します。

Multiplayer Services パッケージは、ゲームにオンラインマルチプレイヤー要素を加えるためのワンストップソリューションです。UGS を利用して、Relay や Lobby などのサービスの機能を 1 つに統合し、新しい "Sessions" システムとして提供されます。これにより、プレイヤーグループのつながり方をすばやく定義できます。

Multiplayer Services パッケージを使用すると、ピアツーピア (P2P) セッションを作成しながら、複数の方法でプレイヤーをそのセッションに参加させることができます。例えば、参加コード、アクティブなセッションのリスト参照、”Quick Join” などです。

次のステップ

ネットワーキングの概念と Netcode for GameObjects の実践的な経験に関する確固たる基礎ができたところで、マルチプレイヤー開発の学びをさらに深めていきましょう。

Unity Learn を利用してみる : マルチプレイヤー開発の学習カーブをさらに緩やかにできるよう、新しいガイド付きコンテンツを用意しています。Unity Learn の最初のマルチプレイヤーコースでは、Netcode for GameObjects の基礎を学習し、初めてネットワークを介したプロジェクトに取り組む方法をガイドします。

サンプルプロジェクトを探る :『Boss Room』、『Galactic Kittens』、『Bitesize Samples』などのサンプルプロジェクトを掘り下げてみましょう。これらのプロジェクトに手を加えると、新しい手法がわかり、ネットワークを介したマルチプレイヤー開発に取り組む際のベストプラクティスを学習できます。

マルチプレイヤーサービスを実験する : 一から自分で作る必要はないことを覚えておいてください。Relay、Lobby、Matchmaker は、プロジェクトにすぐにデプロイでき、開発時間を大幅な短縮につながります。

上級者向けの手法をマスターする : プロジェクトのニーズが増大するにつれ、Netcode for Entities の活用を検討しましょう。この上級者向けのパッケージには、高度なネットワークパフォーマンスを実現するためのクライアントサイド予測やラグ補償などの手法が含まれています。『Megacity Metro』と『ECS Racing』のサンプルでは、これらの機能のアクションを紹介しています。

楽しいネットワーク開発になりますように。


Unity®