Introduction

Introduction

If you want to get started immediately, read the Overview chapter in its entirety, followed by the Transport and Session pages in the Modules chapter, before proceeding to the Tutorials.

This documentation focuses on explaining concepts, architecture, and usage. For detailed API references, consult the included XML documentation in the source code.

The sections progress gradually from theory to implementation:

- **About**
  - Introduction - outlines the structure of the documentation.
  - Background - reviews prerequisite C#, Unity, and networking fundamentals.
- **Content**
  - Overview - presents a conceptual description of the framework's architecture.
  - Modules - describes each framework component, organized by module.
  - Included - explains the included implementations used by the tutorials.
  - Tutorials - provides step-by-step guidance for performing common tasks.

Background

Background

This chapter reviews C# features, Unity constraints, and fundamental networking concepts. It is a refresher on concepts relevant to the framework; it is not a comprehensive primer.

Topics include:

- **C#**
  - Memory-efficient, stack-only types
  - Multithreading and synchronization
  - Partial classes
- **Unity**
  - Assembly definition files (asmdefs)
  - Assembly references (asmrefs)
  - Multithreading within Unity
  - Partial classes within Unity
- **Networking**
  - Core concepts (packets, sockets, protocols, etc.)
  - Transmission concerns (Latency and packet loss)

C#

C# Fundamentals

Memory and Structs

In C#, value types such as structs are stored directly on the stack or inline within other objects, making them faster to allocate and cheaper to copy than reference types. However, careless use of large structs can lead to unnecessary copying. Immutable structs can reduce unnecessary copying and make it easier for the compiler and JIT to optimize code.

Marking a struct as `readonly` ensures that all of its fields are immutable once constructed. This prevents accidental modification and enables the compiler and JIT to perform optimizations such as passing the struct by reference. Immutable structs also simplify reasoning about thread safety, since they can be freely shared across threads.

Ref structs are stack-only types that cannot be boxed, captured by lambdas, or stored on the heap. This restriction makes them ideal for representing temporary views of memory that should not escape the current scope. They are often used to improve performance and reduce allocations by operating directly on existing memory buffers.

The `stackalloc` keyword allows developers to allocate memory directly on the stack, providing a fast, temporary buffer whose lifetime is scoped to the containing method. Unlike heap allocations, `stackalloc` does not incur garbage collection overhead, making it ideal for short-lived arrays or buffers, such as those used for packet serialization or processing.

`Span<T>` is a ref struct that provides a safe, efficient reference to a contiguous region of memory, whether it comes from an array, stack allocation, or unmanaged buffer. `ReadOnlySpan<T>` is the immutable counterpart, used when the indices should not be modified. Both are stack-only, they avoid garbage collection pressure while still providing bounds checking and type safety. Unlike arrays, spans do not require copying or new allocations when slicing.

Threading and Synchronization

Networking systems often perform multiple operations concurrently, such as sending, receiving, parsing, and dispatching packets. Understanding how threads and synchronization work in C# is essential for writing safe and efficient code.

A thread is an abstraction representing a set of instructions that a CPU core can execute. Multiple threads can execute concurrently, either in parallel on different CPU cores or via time-slicing on the same core, but managing their execution must be done carefully to avoid performance issues.

When multiple threads access shared resources, synchronization is required to prevent race conditions. Threads often need mechanisms to safely coordinate access to shared resources. The `lock` keyword enforces mutual exclusion, ensuring only one thread executes a critical section at a time. `ThreadLocal<T>` allows each thread to maintain an independent instance of a variable. For low-latency scenarios, atomic operations via the `Interlocked` class and lock-free structures like `ConcurrentQueue<T>` can be used instead of traditional locks.

Partial Classes

Marking a class as `partial` allows it to be split across multiple files. At compile time, the C# compiler combines all partial definitions into one complete class. This allows additional functionality to be added in separate files without modifying the original source file.

Unity

Unity

Multithreading

Unity enforces a single-threaded execution model for most engine APIs. Threads running outside the main thread cannot safely access or modify gameobjects, components, or the scene. All code that reads from or modifies gameobjects, the scene, or Unity components must run on the main thread.

A `SynchronizationContext` provides a mechanism to schedule work back onto the main thread, ensuring that code interacting with Unity APIs executes in a safe context. Background threads can post work to the main thread through a `SynchronizationContext`.

Developers must ensure that all interactions with Unity objects from background threads are processed through the main thread to avoid undefined behavior or crashes.

Partial Classes

Assembly definition files (asmdefs) allow developers to create separate assemblies within a Unity project. By default, Unity compiles all scripts in a folder into a single assembly. An asmdef applies to all scripts in its folder and any subfolders, enforcing compiler-level boundaries and giving developers explicit control over how code is grouped.

Assembly references (asmrefs) declare dependencies between assemblies and provide a means by which an asmdef can be accessed by classes outside of its scope. When one assembly requires access to types in another assembly, an asmref permits the compiler to recognize the relationship.

Within Unity, asmrefs are necessary to allow external scripts to access or extend partial classes that are defined in another assembly. By referencing the framework's asmdef, asmrefs enable developers to extend its partial classes from external assemblies.

Networking

Network Fundamentals

Endianness

Endianness is the order in which bytes are arranged in memory. The same information can be stored differently on big-endian and little-endian systems, which can lead to errors if two systems interpret the bytes differently. Specifically, endianness describes whether the byte containing the most significant bits of a multi-byte value is stored at the lowest or highest memory address.

For modern development, this is rarely a concern, but it can matter when working with specialized hardware or network protocols that require a specific byte order. All major modern desktop, mobile, and console platforms in active use today are little-endian, though some historically used big-endian architectures and a few operating systems still technically support big-endian.

Packet

Before data can be transmitted over a network, it must be serialized into a sequence of bytes and stored in a buffer. Protocols are contracts that define how buffers are packaged, interpreted, and exchanged. A packet is a transmission unit that encapsulates a buffer with protocol-specific metadata.

Network protocols create packets by adding headers used for routing and delivery. Application protocols may mimic this behavior to manage buffer exchanges independently of the network's transmission mechanisms.

At any given layer, the payload refers to the data produced by the layer above, while the additional information introduced by the current layer implements that layer's protocol. A buffer may contain both the payload from the layer above and the protocol information added by the current layer.

Socket

A socket is an endpoint for sending or receiving packets on a network. It acts as a communication link between two programs, typically identified by an IP address and a port number. Sockets provide an interface to the underlying operating system's networking stack.

Socket Address

An IP address identifies a device on a network. There are two common versions: IPv4 (e.g., 192.168.1.1) and IPv6 (e.g., 2001:0db8::1). IP addresses can be public, routable over the internet, or private, used only within local networks.

A port is a 16-bit number (ranging from 0 to 65535) that identifies a specific process or service on a device. Ports are categorized into well-known ports (0-1023), registered ports (1024-49151), and dynamic or private ports (49152-65535). Well-known ports are reserved for standard services like HTTP (port 80) or FTP (port 21). Dynamic ports are typically used for temporary or private connections.

Together, an IP address and port form a socket address, allowing multiple services to run on the same device without interfering with each other. While IP addresses map to network interfaces at the hardware level, ports serve as an abstraction to differentiate multiple sockets on the same interface.

In .NET, the Endpoint class is an object that represents a socket address, providing an API to access the IP address, port number, and address family (e.g., IPv4 or IPv6). Endpoints abstract the details of the underlying network interface, allowing developers to work with connections in a platform-independent way.

Firewall and NAT

IP addresses can be public, meaning routable over the internet, or private, only existing within a local network. Most devices on a home or office network are assigned private IP addresses by a router or network gateway. These private addresses are not directly reachable from the internet. Instead, the router has public IP addresses that represent the local network externally. Through a process called Network Address Translation (NAT), the router forwards incoming internet traffic for the public IP addresses to the appropriate private IP addresses inside the network by associating a unique port with each device. This allows multiple devices with different private IP addresses to share a single public IP address when communicating with external servers.

Firewalls are network security systems that monitor and control incoming and outgoing traffic based on predetermined rules. They can block or restrict certain types of traffic, prevent unauthorized access, and help protect devices from attacks. Firewalls can operate on individual devices, on routers, or at the network perimeter, and they can affect socket communication by blocking ports, IP addresses, or protocols. Combined with NAT, firewalls may require special handling to allow direct connections between devices across networks.

Firewalls and NAT devices typically allow inbound traffic only if it matches an existing outbound connection, meaning unsolicited incoming packets are usually blocked. To enable connections between devices behind such protections, applications often use publicly reachable servers as intermediaries.

A common technique for enabling direct peer-to-peer communication is hole punching. Each peer first sends outbound packets to a known server so that the NAT/firewall opens a temporary mapping. The server then informs each peer of the other’s public IP address and port, allowing them to exchange packets directly. Importantly, some routers only check whether an incoming packet's destination IP address matches an existing mapping and do not verify that the packet’s source IP address and port match the original address that created the mapping. In such cases, as long as a packet fits a known mapping, it is forwarded to the correct device.

While hole punching often works for typical NATs and firewalls, it is not guaranteed to succeed in all network configurations. Some NAT types, such as symmetric NATs, or strict firewall rules can prevent peers from establishing a direct connection, even if they know each other's public IP address and port. In these cases, fallback strategies become necessary to maintain reliable connectivity. One common approach is relaying traffic through a public server, which forwards packets between peers when direct communication is blocked. Another option is explicit port forwarding, where the user manually configures the router to allow incoming connections to reach a specific internal device.

UDP and TCP

UDP (User Datagram Protocol) is a connectionless socket protocol. It sends discrete packets without establishing a persistent connection and does not guarantee delivery, ordering, or protection against duplication. UDP is lightweight and fast, making it suitable for real-time applications such as games or voice communication, where latency matters more than reliability.

TCP (Transmission Control Protocol) is a connection-oriented protocol that establishes a session before data transfer. It transmits data as a continuous, ordered stream of bytes rather than discrete messages. Beyond the initial handshake, each portion of the stream carries additional overhead, including acknowledgments, sequence numbers, and congestion control. These extra packets and processing steps increase latency and bandwidth usage, which can make TCP less suitable for applications where immediate delivery is more important than guaranteed reliability.

MTU and Fragmentation

The Maximum Transmission Unit (MTU) is the largest size, in bytes, that a single packet can be when sent over a network. Fragmentation occurs when a large payload is divided into smaller payloads to fit within the network's MTU. Each fragment is sent separately and reassembled at the destination.

Fragmentation increases overhead and can lead to delays or even packet loss. Each fragment requires its own headers, and if any fragment is lost, the entire original packet must be resent. Excessive fragmentation can reduce network efficiency and increase latency, particularly on unreliable or congested networks.

For most networks, the typical MTU is 1500 bytes. To avoid fragmentation, it is recommended to keep the payload size smaller than this, usually around 1400 to 1450 bytes to account for headers added by socket protocols. By staying under this size, packets are less likely to be fragmented, which improves performance and reliability.

Latency and Jitter

Latency is the time it takes for packets to travel from a sender to a receiver, usually measured in milliseconds. It represents the delay in communication over a network, and high latency can make interactions feel delayed or unresponsive.

Latency is not always equal in both directions, meaning the time it takes for a packet to travel from sender to receiver can differ from the return trip. Ping is a common way to measure latency, showing how long it takes for a packet to travel to an endpoint and back.

Jitter is the variation in latency between packets sent from a sender to a receiver. Even if the average latency is low, fluctuations in delivery times can make a network connection feel unstable.

Packet Loss

Packet loss is when one or more packets fail to reach their destination. This can occur due to network congestion, faulty hardware, or unreliable connections.

Overview

Overview

This chapter presents a conceptual overview of messages, protocols, and routers, with analogies to clarify each concept. Its purpose is to provide a foundation for understanding the concepts rather than detailing their concrete implementations.

- A message is a local transmission unit for buffer exchange.
- A protocol is a contract that defines a process for exchanging messages.
- A router is a hop that directs a message's path through an application.

Routers are connected by protocols to form a hierarchy of parent-child relationships, where inbound messages flow from parent to child and outbound messages flow from child to parent. Protocols serve as the mechanism by which messages are exchanged between routers.

Message

Message

A message is a local transmission unit that packages a buffer with metadata. A buffer is a container for serialized data, while metadata provides information about a message itself. Metadata can be packed within a buffer or travel alongside it.

A route is metadata that denotes a channel and an endpoint. A channel represents a relative hop in a routing hierarchy. An endpoint specifies a message's source or destination depending on whether the message is inbound or outbound.

For an inbound message, the channel is extracted from the message’s buffer to determine the route. The channel represents the next hop toward the message’s destination. The endpoint represents the source of the message’s buffer.

For an outbound message, the channel is injected into the message’s buffer to denote the route. The channel represents the previous hop away from the message’s source. The endpoint represents the destination for the message’s buffer.

Example Analogy

Imagine two identical neighborhoods with layouts that are mirror images of each other, called Inboundville and Outboundville. Each house in the neighborhoods can send envelopes to its mirrored instance in the opposite neighborhood.

An envelope represents a message, and the contents inside represent its buffer. Each house is a hop, and the paths connecting them represent channels. The neighborhoods themselves represent endpoints.

The houses are organized in a hierarchy, where they act as gatekeepers (parents) that route envelopes to their interior (child) houses, which may themselves have interior houses. This creates a tree-like structure, with envelopes flowing through multiple levels of the hierarchy before reaching their destination.

Consider an example in which a house in Outboundville sends an envelope to its mirrored instance in Inboundville. As the envelope is routed toward the exit of Outboundville, each house adds a post-it note to the letter inside the envelope, representing the channel metadata for that hop.

Each post-it note records the previous path the envelope took to reach the current house. Over time, these post-it notes accumulate on the letter, layer by layer, forming a stack that contains the full path back to the source house.

When the envelope arrives at Inboundville, it is opened and the top post-it note on the letter is removed to determine the next path on the route. At each subsequent house, this process is repeated. By removing the post-it notes in the reverse order they were added, the mirrored route is retraced through Inboundville until it arrives at the correct house.

Protocol

Protocol

A protocol is a contract that defines a process for exchanging messages. Protocols execute this procedure uniformly, serving as context-agnostic mechanisms for managing message exchange.

Protocols can be bound together to form recursive chains of behavior, with messages passing from one protocol to the next until a final message is produced. Protocol chains present as black boxes that process messages sequentially based on whether a message is inbound or outbound.

A protocol may:

- Compress or decompress message buffers.
- Ensure received messages are output in the correct sequence.
- Deconstruct message into and reconstruct them from fragments.

Despite being context-agnostic, protocols can be stateful and may invoke protocol-related events. Protocols can also expose variables that are adjustable at runtime to control behavior, such as the number of times a message is transmitted or the interval between transmission attempts.

Example Analogy

Recall the analogy from the previous page. In both neighborhoods, houses don't exchange envelopes directly. Instead, the exchanges are handled by machines called protocols, which take envelopes as input and produce new envelopes as output according to a set of rules. A single envelope might be split into several, or multiple envelopes might be combined into one.

Imagine one of the houses has a machine that encrypts or decrypts envelope contents depending on whether the envelope is outbound or inbound. When an outbound envelope is placed inside the machine, it produces a new envelope whose contents are a coded version of the original, which only an identical machine on the mirrored house in the opposite neighborhood can decipher.

When an inbound envelope arrives at the mirrored house, it is placed into an identical machine. The contents of the envelope appear unreadable, but the mirrored machine decodes the sequence of symbols and produces an envelope whose contents are identical to the original.

Some houses might use multiple machines in sequence, each feeding their output into the next machine. To reconstruct the original contents, the sequence to process inbound envelopes is the reverse order of the outbound sequence. Each house must use the same machines, arranged in the same order as its mirrored counterpart; otherwise, envelope contents may be corrupted.

Router

Router

A router is a hop that directs a message's path through an application. Routers form hierarchies of parent-child relationships that messages traverse. Each router provides channels, which child routers can register their protocols to, allowing messages to be exchanged between them.

Routers are context-aware, enabling sophisticated routing and filtering behavior. Context refers to auxiliary information that accompanies a buffer but is not transmitted over the network.

Example Analogy

In the previous examples, hops were represented as houses. A router is a hop capable of adaptive routing. It’s more accurate to think of them as households with people. The people can inspect documents attached to envelopes, communicate with other households, or observe the state of the neighborhood, allowing them to make decisions based on context.

Unlike the protocol machines, which always process envelopes the same way, the people in a household can decide how to handle each envelope individually. They might batch several together, prioritize forwarding specially marked envelopes, or even hold some back based on their importance and the current load of envelopes in transit.

Each household can route envelopes according to its own judgment, creating a flexible, context-aware flow of information through the neighborhood.

Modules

Modules

This chapter describes the components used to implement the concepts presented in the preceding Overview chapter.

The pages are organized from the high level, Application and Session, modules to the low level Transport and Utilities, modules but can be read in any order. The Utilities page describes supporting components, that can be reviewed as needed while reading the other pages.

For a quick start, it is possible to proceed directly to the tutorials. However, it is recommended to review the Application and Protocol pages, as these provide essential context for understanding the upcoming sections.

Application

Application

The application module provides high-level components for creating and routing messages within an application and across a network.

Router

`Router` is the base component responsible for directing the flow of messages through an application. Routers form hierarchical parent and child relationships. Parents provide channels that their children can register protocols to. These registrations determine how messages traverse a hierarchy.

Routers are context-aware, allowing them to route messages based on their metadata. This enables custom routing, filtering, prioritization, and modification of messages based on operational conditions and application state.

As the foundation for higher-level components such as `Dispatcher` and `Node`, it establishes the core mechanisms for message flow within the framework.

Node

`Node` is a router specialized for concurrent socket operations. It can run dedicated worker threads to process socket operations, supporting both IPv4 and IPv6 communication. To support concurrent execution, nodes maintain a pool of managed objects for storing arguments associated with queued socket operations.

It supports dual-mode socket operations with automatic address-family remapping and can bind to distinct IPv4 and IPv6 endpoints simultaneously. Nodes may also be rebound at runtime, allowing endpoints to be released and reattached without reconstructing the node.

Dispatcher

`Dispatcher` is a router specialized for serialization, serving as a source and destination for messages. Dispatchers construct messages with routing information and optional contextual data. Dispatchers implicitly serialize and deserialize unmanaged types for efficient transmission.

Session

Session

The Session module provides components for message processing and session management. A message bundles a buffer with a route and optional contextual information. A packet associates a message with a header. Protocols use the combination of a packet's route and its header to manage protocol-related session state across multiple operations.

Message

`Message` provides a shared abstraction for routers and protocols to exchange buffers. The generic form, allows components to work directly with a typed context, automatically converting between a span of bytes and an unmanaged struct. This design enables custom components to operate with application-specific context without breaking components that operate on the non-generic form.

Route

`Route` encapsulates metadata that Routers use to direct messages through a hierarchy. `WeakRoute` is an unmanaged variant that holds a weak reference to its endpoint, allowing the endpoint to be garbage collected when no longer in use.

Protocol

`Protocol` serves as a base abstraction for implementations of `IProtocol`. It handles chaining, callback routing, and method propagation, enabling derived types to define protocol behavior.

Consumers are responsible for calling `Dispose()` to free associated resources.

Packet

`Packet` encapsulates a message with additional information required by protocols to appropriately process the associated buffer. `Header` contains session-related information for packets, including both a type and numeric identifier. Headers allow protocols to differentiate and track messages via a lightweight and standardized structure.

Cache

`Cache` is a specialized collection for temporary packet storage. It retains packets and messages by their route and header, enabling packets to persist beyond their normal lifetime of ref structs and be accessed later as needed.

Consumers are responsible for calling `Dispose()` to free associated resources.

Transport

Transport

The transport module provides low-level abstractions for managing network endpoints and sockets in a high-performance, allocation-free manner. The API mirrors the .NET networking API for familiarity with additional convenience enhancements. NetSocket, NetEndpoint, and NetSocketAddress wrap or replace standard .NET networking classes to reduce overhead and minimize allocations, providing an efficient foundation for high-performance networking components.

NetSocketAddress

`NetSocketAddress` encapsulates information that defines a network endpoint. Designed to support efficient reuse and manipulation, they can be created from a span of bytes, written to a span, and remapped between IPv4 and IPv6 without creating new objects.

NetEndpoint

`NetEndpoint` encapsulates a socket address with a thread-safe, memory-efficient API for instance creation. This class ensures that duplicate endpoint objects are avoided, provides safe concurrent access, and allows endpoints to be stored and shared efficiently throughout an application.

NetSocket

`NetSocket` provides a unified interface to manage an underlying system socket. It accesses platform-specific socket APIs directly to minimize memory allocations and improve efficiency. When bound to a NetEndpoint, a NetSocket can perform zero-allocation UDP send and receive operations.

Utilities

Utilities

The utilities module provides a set of tools and helper components for efficient, allocation-free operations.

NetInfo

`NetInfo` provides static methods for inspecting and retrieving local network details.

WeakDictionary

`WeakDictionary` is a dictionary-like collection that holds weak references to its keys and/or values, allowing them to be garbage collected when no longer in use. As long as the referenced objects remain in memory, the collection provides normal key–value access. Once those objects become unreachable elsewhere, they are automatically removed, reducing object churn and memory pressure by reusing existing objects without extending their lifetimes.

Consumers are responsible for calling `Dispose()` to free associated resources.

SpanExtensions

`SpanExtensions` provides a set of helper methods perform allocation-free manipulation of spans, lists, and dictionaries. It enables precise control over memory and layout, allowing transformations and operations to be performed directly in-place with predictable performance and minimal overhead.

Scheduler

`Scheduler` ensures callbacks run at consistent intervals on the appropriate thread, providing reliable timing without manual thread management. Callbacks are dispatched safely, allowing concurrent execution without blocking other operations.

Consumers are responsible for calling `Dispose()` to free associated resources.

Schedulers integrate with Unity to maintain consistent timing during editor pauses.

Chronometer

`Chronometer` calculates an elapsed time relative to a reference point. It measures durations reliably even if the system clock changes, ensuring consistent timing for operations that require fine-grained temporal control.

Included

Included

This section details the components used in the upcoming Tutorials section. Pages that cover an abstract component include an expandable description for its included implementation. Except where stated otherwise, the included implementations are suitable for performance critical environments.

Prerequisite Material:

- Message
- Protocol
- Router
- Application
- Session
- Transport

Implementations often share a name with their abstract component, which may be named after a related concept. To avoid confusion, types, members, parameters, values, and keywords are highlighted, while concepts are written as plain text.

For example, `Protocol` refers to the included implementation of the abstract `Protocol<T>` class, whereas protocol refers to the concept of a message-processing unit.

The term *internal* refers to code contained within the framework's assembly, whereas *external* refers to code in separate assemblies, including any included implementations.

Protocol

Protocol

`Protocol<T>` is an abstract implementation of the `IProtocol<T>` interface, constrained to `Enum` types.

`Protocol<T>` instances can be bound together via `Bind(Protocol protocol)` to create multi-stage processing chains. The sequence of protocol executions is fixed for a given chain, but the order of execution depends on whether a message is inbound or outbound.

`Bind(Protocol protocol)` returns the `protocol` argument after it has been bound. The returned instance acts as a façade for the entire chain. Method calls on a chain's leaf node are propagated through the entire chain.

`Run()` calls `Execute()` on every protocol in a chain; however, a protocol can call `Execute()` internally without propagating the call through the rest of the chain.

`Clear()` is used to reset the state of a `Protocol<T>` chain, whereas `Clear(Route route)` is used to exclusively clear the state associated with a specific route. Internally the logic is defined within `OnClear()` and `OnClear(Route route)`.

Inbound messages enter a chain via `Receive(Message message)` and exit via the `Inbound` event.

`InvokeInbound(Message message)` emits a message by passing it to the next protocol in the chain; if none exists, it invokes the `Inbound` event.

Outbound messages enter a chain via `Send(Message message)` and exit via the `Outbound` event.

`InvokeOutbound(Message message)` emits a message by passing it to the next protocol in the chain; if none exists, it invokes the `Outbound` event.

Event handlers can be added via `Register(T e, ProtocolEventHandler handler)` and removed via `Unregister(T e, ProtocolEventHandler handler)`.

Consumers are responsible for calling `Dispose()` to free associated resources.

Protocol

The included `Protocol<ProtocolEvents>` implementation defines a constructor, two fields (a nested `Configuration` class and a `SynchronizationContext`), and a static factory method.

`ProtocolEvents` defines events for which custom protocols can invoke registered callbacks.

`New(List<Protocols> protocols, Configuration config, SynchronizationContext context)` constructs a protocol chain from a list of `Protocols` and returns the leaf node.

The `protocols` argument is used to create and bind a sequence of `Protocol` instances. Each `Protocols` value corresponds to a `Protocol` implementation. `Protocol` instances are created in sequence, starting at `protocols[0]`. Each `Protocol` instance created is bound to the next, forming a chain. The final instance is returned to the caller; it represents the entire chain.

The `config` argument is an instance of the nested `Configuration` class, supplied by the caller to allow the returned `Protocol` instance to reference configurable values.

The `context` argument provides a mechanism for the returned `Protocol` instance to post work to the specified thread.

The `Protocol(Configuration config, SynchronizationContext context)` constructor creates and returns a single instance of `Protocol`.

> Inbound messages enter a chain at the lowest index and exit from the highest index.
>
> Outbound messages enter a chain at the highest index and exit from the lowest index.

Unreliable

Unreliable

`Unreliable` is the simplest possible implementation of `Protocol`. `Unreliable` does not modify inbound or outbound messages in any way. It can serve as a placeholder wherever an instance of `IProtocol` is required.

The constructor requires the same parameters as the base `Protocol` constructor, but it does not use them. It passes both values to the base constructor and performs no additional setup.

Reliable

Reliable

`Reliable` extends the included `Protocol` class to demonstrate how to implement reliable and ordered message exchange. It handles scheduling, ordering, synchronization, and recovery. It is not suitable for performance critical environments; however, its inefficiencies can be mitigated if outbound messages are batched per-endpoint prior to network transmission.

Unique IDs are designated sequentially, starting at zero, for each payload. A packet's ID corresponds to the payload it references. The protocol utilizes the following packet types, denoted by a packet's header:

- `DAT` - A packet containing a payload. A copy of each outbound `DAT` packet is stored locally until an acknowledgement is received or its route is cleared.
- `ACK` - An acknowledgment that a payload was received. Once an `ACK` is received, the local copy of its corresponding `DAT` packet is discarded.
- `REQ` - A request for a payload to be re-transmitted. A `REQ` is sent when a `DAT` packet is received ahead of schedule, based on its ID. If a `REQ` references a payload that was already acknowledged, a `CLR` packet referencing the most recently acknowledged payload will be sent as a reply to verify the sessions are synchronized.
- `CLR` - A check to verify synchronization or initiate a recovery process. If a `CLR` packet is received that references a payload that has not yet been received, the sessions are out of sync. To recover, a `CLR` packet with an ID of zero will be sent as a reply and the route will be cleared. When a `CLR` packet with an ID of zero is received, the corresponding route is cleared.

`Configuration` defines the following members used by the Reliable protocol:

- `ReliableRecvBuffer` - Denotes the number of packets that can be buffered ahead of the next expected packet, based on id, prior to its receipt.
- `ReliableRecvAttempts` - The maximum number of times the `ReliableRecvInterval` may elapse while a given packet is stored in the `inbound` cache before the packet is discarded.

- `ReliableRecvInterval` - The period of time that must elapse before the number of receive attempts associated with a given packet is incremented.
- `ReliableSendAttempts` - The number of times the `ReliableSendInterval` may elapse while a packet is stored in the `outbound` cache until the `Unacked` callback is invoked.
- `ReliableSendInterval` - The period of time that must elapse between send attempts for a packet, stored in the `outbound` cache.

The `Packet(Header header, Message message, Span<byte> buffer)` constructor is used to create outbound packets, as it injects the `header` and message buffer into the packet's contents.

The `Packet(Message message)` constructor is used to create inbound packets, as it extracts the `Header` from the message's buffer and leaves the remaining portion of the buffer as the packet's `Contents`.

`OnOutbound(Message message)` sends outbound messages and schedules the re-transmission of unacknowledged outbound messages. It uses the `outbound` cache to store local copies of outbound packets for re-transmission.

`Inbound(Message message)` processes inbound messages based on their type. It passes payloads to `OnData(Packet packet)`, which may process them immediately or store them in the `inbound` cache if they arrive ahead of schedule.

`In()` and `Out()` are each executed by a Scheduler. Both increment a counter for each packet in a cache if a configured time interval elapses. Each counter continues to increment until it reaches a configured limit, at which point a corresponding action is triggered. If the limit is reached, `In()` removes the packet from the `inbound` cache, whereas `Out()` invokes the `Unacked` callback. If the callback does not clear the packet's route, its attempt counter is reset to zero.

`Execute()` calls `Out()` to enable consumers to manually resend unacknowledged outbound packets.

Example Analogy

Imagine two post offices, Post Office A and Post Office B, that coordinate to reliably deliver letters and process them in order. All exchanges are done using envelopes, each containing a label with a unique identifier and a classification code.

Identifiers are assigned sequentially, starting at zero, for each letter. The classification code indicates the purpose of the envelope: carrying a letter, acknowledging receipt, requesting a missing letter, or synchronization exchanges.

Post Office A keeps a copy of every letter it sends to Post Office B. When a letter is received at Post Office B, Post Office B immediately sends back an acknowledgment. Upon receipt of the acknowledgement, Post Office A discards its copy of the letter.

If no acknowledgment is received, Post Office A re-sends the letter as specified by its policy, which defines how many times to attempt delivery and the minimum time that must elapse between attempts. If the letter has not been acknowledged after the attempts, the sender is notified. If the sender does not cancel the delivery, Post Office A will repeat the process.

Any letters that arrive at Post Office B ahead of schedule are held in a staging area. Post Office B requests missing letters from Post Office A. Staged letters are processed, in order, as missing letters are received. If Post Office A receives a request for a letter that Post Office B already acknowledged, it indicates that the post offices are out of sync.

If this occurs, Post Office A can send an envelope to Post Office B containing the identifier of the last acknowledge letter it received from Post Office B. If this identifier is higher than the identifier of any letter Post Office B has received from Post Office A, Post Office B clears its assigned identifiers and instructs Post Office A to do the same.

By following these procedures, both post offices remain synchronized, ensuring that letters are reliably delivered and processed in order.

Router

Router

The framework's `Router` definition is incomplete. The class is defined as `partial` to allow it to inherit from classes that are not defined within the framework. However, `Router` needs access to an instance of `Router.Args` to perform initialization.

A second partial implementation must define an `Initialize()` method that returns an instance of `Router.Args`. This enables `Router` to inherit from external classes while centralizing initialization for its subclasses.

If `Router` relied on an abstract method to return an instance of `Router.Args`, each subclass, including `Dispatcher` and `Node`, would need to define a separate implementation. To avoid this, `Router` relies on an undefined `Initialize()` method that returns an instance of `Router.Args`.

The `Router.Args(byte channel, Router parent, IProtocol protocol)` constructor requires three arguments:

1. `channel` - In `Router` hierarchies, parents extract channels from inbound message buffers and insert channels into outbound message buffers. Child routers register their protocol to a channel via their parent's `Register(byte channel, IProtocol protocol, bool replace = false)` method. Routers can undo this via the `Unregister(byte channel, IProtocol protocol)` method.
2. `parent` - The router's parent in the hierarchy.
3. `protocol` - The `IProtocol` instance, that will connect the router to its parent.

`Router` properties are lazy-loaded via `Initialize()` the first time any public member is accessed.

`Enable` and `Disable` manage protocol registration, skipping it if a router is at the top of the hierarchy.

`OnOutbound(IProtocol protocol, Message message)` extracts route channels from message buffers and can be overridden to control the metadata that is extracted from inbound message buffers.

`OnOutbound(IProtocol protocol, Message message)` injects route channels into message buffers and can be overridden to control the metadata that is inserted into outbound message buffers.

`Send(Message message)` and `Receive(Message message)` may be overridden to hook into the routing pipeline after metadata insertion or extraction, but before messages are forwarded to the next hop.

The internal partial definition of `Router` implements `IDisposable`. `OnDisposeManaged()` and `OnDisposeUnmanaged` are defined as partial methods to enable external partial definitions to implement disposal. Subclasses of `Router` can simply override `OnDispose(bool disposing)`.

Router

Although the included `Router` implementation inherits from `MonoBehaviour`, its definition as `partial` necessitates a corresponding non-partial implementation in a separate namespace to be serializable as a Unity component.

The overridable `Configuration` property returns a shared `Protocol.Configuration` instance that is initialized with the default values defined in `Protocol.Configuration`.

`Initialize()`

- Resolves the value of `parent` based on the GameObject hierarchy, falling back to `this` if a parent router is not found.
- Creates `protocol` via the provided `Protocol` implementation's factory method (see Protocol for details).
- Returns a new `Args` object constructed from the serialized `channel`, the resolved `parent` router, and the `protocol` returned from the factory.

To ensure consistent behavior for all Router implementations, `OnEnable()` and `OnDisable()` are kept private and simply call the virtual `Enable()` and `Disable()` methods.

`Log(string message, object obj = null)` is included to consolidate logging behavior for the included `Router` implementation and any of its subclasses.

`OnDisposeManaged()` disposes the underlying protocol. `OnDisposeUnmanaged()` is intentionally not implemented, as it is not required and allows the compiler to ignore the partial declaration in the internal definition.

`OnDestroy()` centralizes resource cleanup for `Router` and its subclasses.

Node

Node

`Node` creates internal `OperationArgs` objects to store arguments for concurrent socket operations. These operation objects allow work to be queued on one thread and then executed on another. A node can run two separate threads, one for send operations and one for receive operations.

`Node` has abstract members that must be implemented:

- `BufferLength` specifies the length of the buffers used by operation objects, rounded up to the nearest power of two (e.g., 26 → 32).
- `MinimizeBuffers` determines whether buffers that exceed the BufferLength will be returned to a pool of buffers so an array with a more appropriate length can be rented from the pool.
- `Error(EndPoint remoteEP, Span<byte> buffer, NodeOperation op, SocketError se)` is called on an operation's thread if the operation fails to complete.
- `Receive(EndPoint remoteEP, Span<byte> buffer, DateTime time)` is called by the framework when `Receive(int count, bool mapIPv6ToIPv4)` is called, to process buffers received on the dedicated receive thread. The `time` argument reflects the value of the virtual `Now` property at the moment the buffer was received.

`OnCompleted(object sender, Operation operation)` is an overridable method called after each successful send or receive operation on the corresponding thread. It invokes the public `Completed` event, with the `operation` argument.

`Prepare(Resources resources)` is an overridable method called after each send or receive operation on the executing thread, and after `Process(int count, bool mapIPv6ToIPv4)`. It provisions objects for concurrent send and receive operations. The `resources` argument, obtained from the `Snapshot` property, provides details related to the node's operation objects.

The snapshot includes:

- `Idle` - The number of objects available for send or receive operations.
- `Engaged` - The number of objects assigned to active send operations.
- `Buffered` - The number of objects committed for receive operations.
- `Pending` - The number of completed receive operations, ready to be processed.
- `Reserved` - The number of times a new operation object was created to execute a send operation, due to a lack of idle objects.

The default implementation of `Prepare(Resources resources)` ensures the idle count is determined with respect to the reserve count and will only provision a new object to be created if all unreserved objects are currently pending. This ensures an object is always available for a receive operation while minimizing the chances the a new object will need to be created for a send operation.

`Provision(int count, int receives)` allocates a number of operation objects equal to `count` and attempts to buffer objects for receive operations from the idle objects, up to the value of `receives`, without regard to the reserved count. To avoid unnecessary allocations, keep the idle objects greater than or equal to the reserved count. Otherwise, an object may be automatically instantiated during a send operation.

Additionally, `Node` includes non-virtual methods for resource management:

`Release(int count, Resource resource)` returns previously provisioned objects to the idle pool, making them available for reuse. Buffered objects are returned up to `count`, reserved objects are decremented (to a minimum of zero), and idle resources are invalid for release.

`Discard(int count, Resource resource)` clears object references, allowing them to be garbage collected. Idle and buffered objects are discarded up to `count`, while releasing reserved objects decrements the reserved count and discards an equivalent number of idle objects.

In addition to defining abstract and virtual members, there are methods that must be called in order for Node to operate:

`Bind(NetEndPoint localEP)` binds a socket to the specified endpoint. A `Node` can be bound to an IPv4 endpoint and an IPv6 endpoint simultaneously. The sockets bound to these endpoints can be accessed via the `IPv4Socket` and `IPv6Socket` properties. Rebinding will replace the socket instance referenced by `IPv4Socket` or `IPv6Socket` based on the address family of the endpoint being bound.

`IPv6Socket` can operate in dual-mode if the feature is supported by the operating system and the socket is bound to an IPv6 address that is IPv4-mappable (e.g., `IPAddress.IPv6Any`). If both an IPv6 and IPv4 socket are bound to an endpoint suitable to receive a packet based on its target socket address, the OS will prioritize the IPv4 socket.

IPv4–IPv6 remapping is required for a dual-mode IPv6 socket to send packets to IPv4 endpoints; however, remapping is optional for received packets. `Send(NetEndPoint remoteEP, Span<byte> buffer, bool mapIPv4ToIPv6)` includes the `mapIPv4ToIPv6` parameter to control remapping. `Process(int count, bool mapIPv6ToIPv4)` includes an analogous `mapIPv6ToIPv4` parameter.

When remapping is enabled, `NetEndPoint.ThreadSafeCreate(SocketAddress socketAddress)` is used to get a reference to each remapped address. If an instance associated with the remapped address already exists, it is returned; otherwise, a new one is created and returned. These instances are tracked by a `WeakDictionary`, and operation objects retain strong references to them as long as possible, to minimize premature garbage collection.

Due to using separate sockets for IPv4 and IPv6 operations, nodes can effectively behave like dual-mode sockets, regardless of whether the featured is supported by the operating system and without the restriction of needing to bind to all interfaces. Each underlying socket of a node can be bound to a distinct IP address and port at the same time.

`Begin(Operations operation, TimeSpan interval)` attempts to spawn a thread for concurrent operations once an endpoint has been bound. The `operation` argument specifies the operation to be executed by the spawned thread. An optional `interval` argument can be provided to control the time between thread executions, reducing computational overhead; if omitted, the thread will block between executions.

`End(Operations operation, TimeSpan timeout)` attempts to terminate the thread corresponding to the specified operation. An optional `timeout` argument can be provided to set the maximum time to wait before abandoning the attempt; if omitted, the method will wait indefinitely. It is recommended to stop provisioning new operation objects while attempting to end a thread to avoid blocking the calling thread indefinitely.

Node

The included `Node` implementation seals all overridable members to improve performance.

`Now` is overridden to return the `Now` property of a `Chronometer` instance to get a more accurate timestamp.

`BufferLength` and `MinimizeBuffers` are set via the inspector.

`Awake()` calls `Bind(NetEndPoint localEP)` to bind both an IPv4 and IPv6 endpoint and sets the `LocalEndPoint` property depending on the value of the `ipv6` serialized field.

`Enable()` calls `Begin(NodeOperation operation)` to start both the send and receive threads and subscribes `Send(IProtocol protocol, Message message)` to its protocol's `Outbound` event to connect outbound messages to the network layer.

`Disable()`calls `End(NodeOperation operation)` to stop both the send and receive threads and unsubscribes `Send(IProtocol, Message message)` from the `Outbound` event to disconnect outbound messages from the network layer.

`FixedUpdate()` calls `Receive(int count, bool mapIPv6ToIPv4)`. `Operations.Pending` is passed for the `count` argument so that all pending receive operations will be processed. `false` is passed for the `mapIPv6ToIPv4` to avoid unnecessary heap allocations.

`Send(IProtocol protocol, Message message)` prepends message buffers with a channel byte to enable receiving endpoints to reconstruct message routes. Route reconstruction ensures that receiving protocols can cache messages by route rather than by endpoint.

`Receive(Endpoint remoteEP, Span<byte> buffer, DateTime time)` extracts a route from the buffer, reconstructs the original message, and then constructs a `Context`, which it packages inside an instance of `Message<Context>`, demonstrating how externally defined context can accompany a message as it is routed through the framework.

Dispatcher

Dispatcher

`Dispatcher` provides methods to streamline serialization and deserialization for messages.

`Endpoint` is an overridable object property used to construct message routes when an endpoint is not provided. The default value is a static object shared by all dispatcher instances.

`OnInbound(IProtocol protocol, Message message)` is sealed and passes message directly to `Receive(Message message)` which can be overridden to implement message processing.

`Read<T>(Span<byte> buffer)` and `Read(Span<byte> buffer, out T value, out int count)` deserialize byte spans, such as message buffers, into unmanaged types.

`Send<T1, T2>(T1 payload, object endpoint, T2 context)` and its overloads use the provided arguments to construct and send a `Message`. The `payload` and `context` parameters are constrained to unmanaged types.

Dispatcher

The included `Dispatcher` implementation extends the base serialization behavior by adding support for reference types via the `SendJSON` and `ReadJSON` overloads.

Objects are encoded as JSON strings, which are then serialized into `Span<byte>` instances.

`Span<byte>` instances can be deserialized into JSON strings and can be used to create a new object or populate the fields of an existing one.

The `log` parameter in the `Send` and `Read` methods specifies whether the generated JSON string is forwarded to the dispatcher's `Log` method. The default value is `false`.

Automatic JSON serialization for reference types is included for debugging and prototyping scenarios. It is not recommended for use in performance-critical environments.

Context

Context

Contextual metadata, that will not be transmitted over the network, may accompany a buffer as it is routed through an application. Such metadata is constrained to unmanaged types to allow for implicit conversions between `Message` and `Message<T>`.

Contextual metadata is useful when information needs to be associated with a message locally. It is particularly helpful for debugging, logging, and time-sensitive operations.

Context

To demonstrate the pattern, the included `Context` struct requires an instance of `DateTime`, to track creation time. `Context` can be extended to include additional unmanaged fields.

Tutorials

This chapter contains practical tutorials that demonstrate the use of the framework's components and extension points. Each tutorial serves as a reference for using or extending the framework, including custom implementations of abstract types.

Included tutorials and topics:

- **Send and Receive Messages**
  Covers: `Span`, `Node`, `Dispatcher`
- **Create a Latency Protocol**
  Covers: `Packet`, `Header`, `Cache`, `Protocol`, `Scheduler`
- **Implement a Synchronous Node**
  Covers: `NetSocketAddress`, `NetEndpoint`, `NetSocket`

Send and Receive Messages

This tutorial demonstrates how to send and receive payloads using the `Dispatcher`, `Message`, and `ProtocolEvents` types.

Create a new `ExampleDispatcher` class that inherits from the included `Dispatcher` implementation and define the following fields:

```
public class ExampleDispatcher : Dispatcher
{
    [SerializeField] string remoteAddress;
    [SerializeField] int remotePort;

    NetEndPoint endpoint;
}
```

Add a `Start()` method that initializes `endPoint` as a `NetEndPoint` created from the `remoteAddress` and `remotePort` serialized fields. Prioritize an IPv4 address if `remoteAddress` cannot be parsed.

```
private void Start()
{
    var ipv6 = NetInfo.GetLocalAddress(AddressFamily.InterNetworkV6);
    var ipv4 = NetInfo.GetLocalAddress(AddressFamily.InterNetwork);
    bool parsed = IPAddress.TryParse(remoteAddress, out var value);
    var address = parsed ? value : ipv4 ?? ipv6;
    var socketaddress = new NetSocketAddress(address, remotePort);
    endPoint = NetEndPoint.ThreadSafeCreate(socketaddress);
    scheduler = new(SynchronizationContext.Current, Execute);
}
```

Both IPv4 and dual-mode IPv6 sockets can receive packets from IPv4 endpoints.

Override the `EndPoint` property to return the newly initialized `endPoint` field. `Send` overloads use the `EndPoint` property if the `endpoint` argument is omitted.

```
protected override object EndPoint => endPoint;
```

Add a serialized `KeyCode` field and an `Update()` method that sends a payload when the specified key is pressed down.

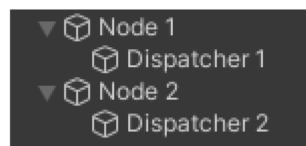
```
[SerializeField] KeyCode send;

private void Update()
{
    if (Input.GetKeyDown(send))
        Send(stackalloc byte[] { 1, 2, 3,});
}
```

> ⓘ `stackalloc` can be used to allocate arrays on the stack to avoid heap allocations.

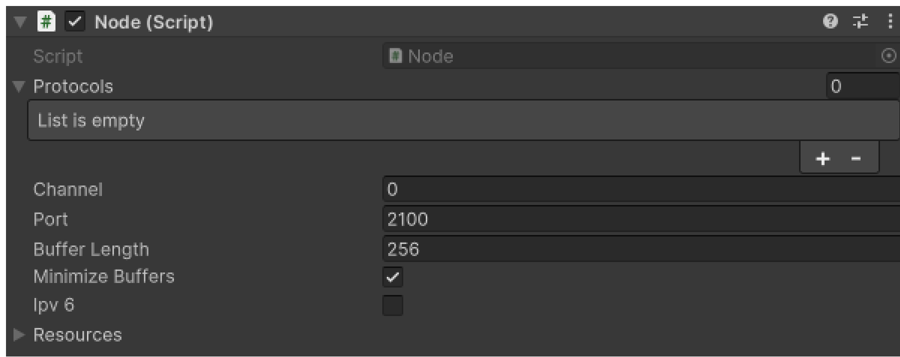
To process a payload after it is received, override `Receive(Message message)`. For inbound messages, `message.Route.EndPoint` represents the source of the payload.

```
protected override void Receive(Message message)
{
    var buffer = message.Buffer.ToDebugString();
    var endpoint = message.Route.EndPoint;
    Debug.Log($"received: {buffer} from: {endpoint}");
}
```

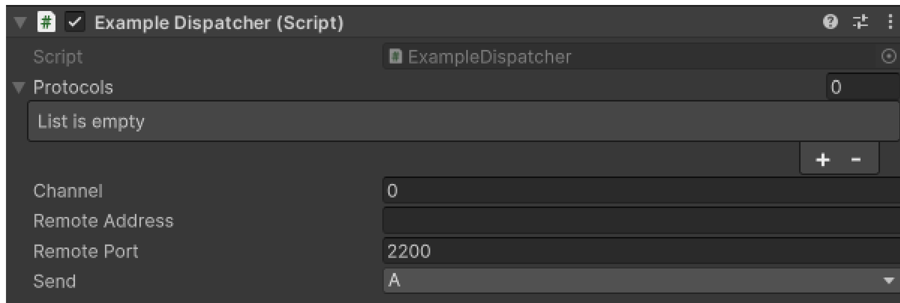
Create two `GameObject` instances, and attach a `Node` component to each. Add a child to each `GameObject` and add the `ExampleDispatcher` component to each child.



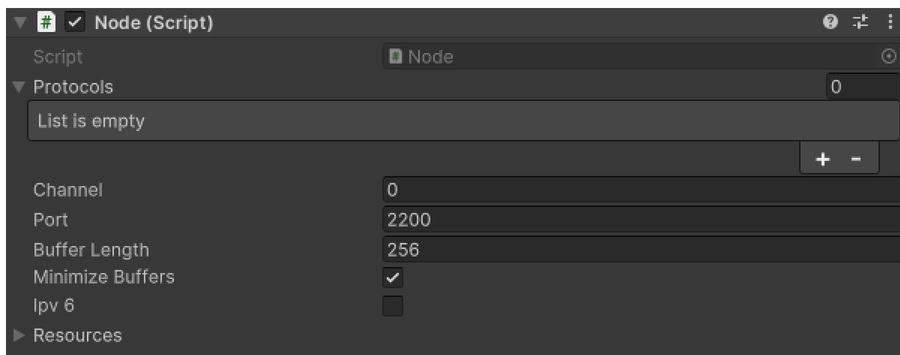
Hierarchy



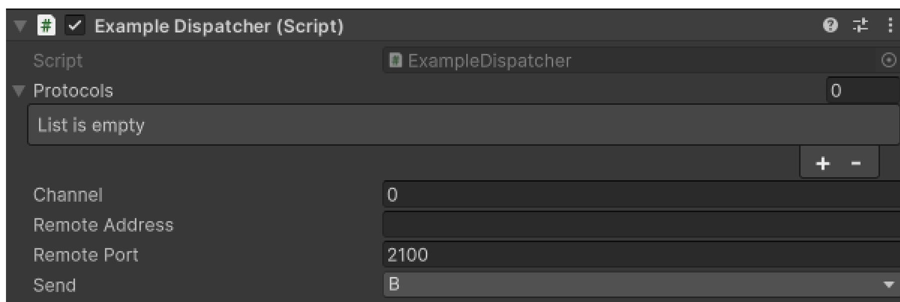
Node 1



Dispatcher 1

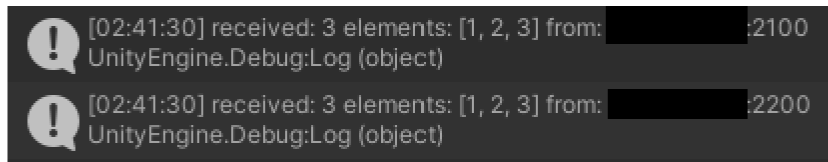


Node 2



Dispatcher 2

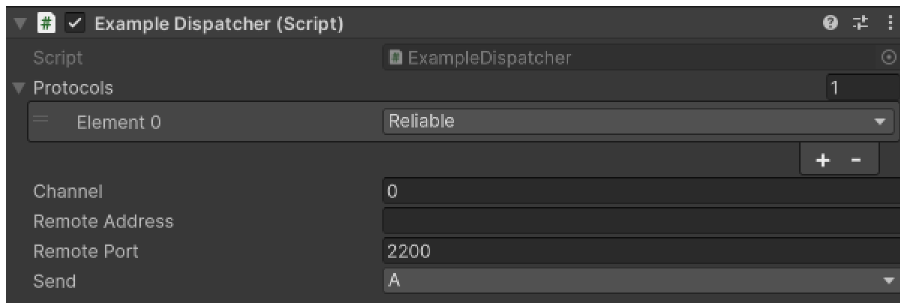
Start the play mode and press the specified keys down. The following debug message should appear:



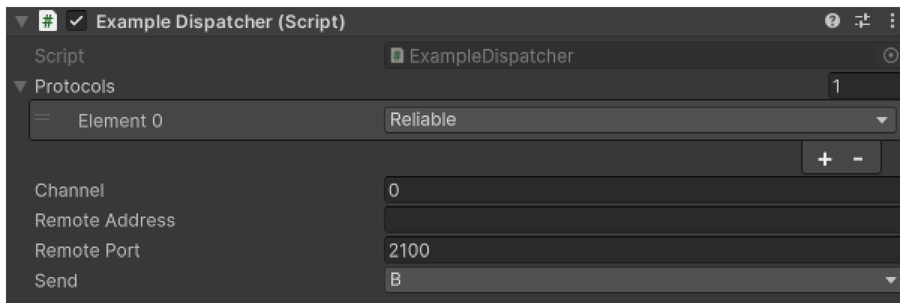
Protocols may raise events and can invoke related callback functions. Register callbacks for the `Acked` and `Unacked` `ProtocolEvents` by overriding `Enable()` and `Disable()`.

```
protected override void OnEnable()
{
    base.OnEnable();
    Protocol.Register(ProtocolEvents.Unacked, UnacknowledgedCallback);
    Protocol.Register(ProtocolEvents.Acked, AcknowledgedCallback);
}

protected override void OnDisable()
{
    base.OnDisable();
    Protocol.Unregister(ProtocolEvents.Unacked,
UnacknowledgedCallback);
    Protocol.Unregister(ProtocolEvents.Acked, AcknowledgedCallback);
}

private void AcknowledgedCallback(IProtocol protocol, Message message)
{
    Debug.Log($"{message.Route.EndPoint} ACKED
{message.Buffer.ToDebugString()}", this);
}

private void UnacknowledgedCallback(IProtocol protocol, Message
message)
{
    Debug.Log($"{message.Route.EndPoint} DID NOT ACK
{message.Buffer.ToDebugString()}", this);
}
```

Add `Reliable` to the `Protocols` list on both Dispatchers.



Dispatcher 1



Dispatcher 2

Debug messages will print to the console when the respective `ProtocolEvents` are raised.

> Try disabling one of the Nodes before sending a packet to trigger `ProtocolEvents.Unacked`.

`Dispatcher` can serialize unmanaged structs implicitly. Add an unmanaged `ExampleStruct` type. Modify `Update()` to send an instance of `ExampleStruct` and modify `Receive(Message message)` to read an `ExampleStruct` from the `message.Buffer`.

```
public struct ExampleStruct
{
    public Vector3 Position;
    public Quaternion Rotation;
}

protected void Update()
{
    ExampleStruct example = new()
    {
        Position = transform.position,
        Rotation = transform.rotation,
    };

    if (Input.GetKeyDown(send))
        Send(example);
}

protected override void Receive(Message message)
{
    if (Read(message.Buffer, out ExampleStruct example))
        Debug.Log($"Position:{example.Position}
Rotation{example.Rotation}");
}
```

Observe the console output from the callbacks to see how the unmanaged struct is represented as bytes.

Reference types must be manually converted. The `SpanExtensions` class contains helper methods to serialize instances of `List<T>` and `Dictionary<K, V>`. Rewrite the dispatcher to send and receive a list of `int` using these methods.

```
[SerializeField] List<int> exampleList;

protected void Update()
{
    if (Input.GetKeyDown(send))
    {
        var size = exampleList.RequiredBufferSizeNoAlloc();
        Span<byte> span = stackalloc byte[size];
        exampleList.PopulateSpanNoAlloc(span);
        Send(span);
    }
}

protected override void Receive(Message message)
{
    var span = message.Buffer;
    exampleList.PopulateFromSpanNoAlloc(span);
    string contents = exampleList.ToArray().AsSpan().ToDebugString();
    Debug.Log($"received: {contents} from: {message.Route.EndPoint}");
}
```

Use the inspector to populate one of the lists. When you send that dispatcher's list over the network you will see the other dispatcher's list populated with identical values.

Alternatively, when performance is not a concern (e.g. during prototyping), objects can be serialized as JSON strings and then encoded as bytes. This has the added benefit of allowing the JSON string to be printed directly to the console for debugging.

The included `Dispatcher` implementation supports JSON serialization for objects via the `SendJSON` and `ReadJSON` overloads. The following example relies on Newtonsoft.Json ↗.

Create an `ExampleClass`.

```
[Serializable]
public class ExampleClass
{
    public string FirstName;
    public string LastName;
}
```

Modify `Update()` to send an instance of `ExampleClass` and modify `Receive(Message message)` to read an `ExampleClass` from the `message.Buffer`.

```
[SerializeField] ExampleClass exampleClass;

protected void Update()
{
    if (Input.GetKeyDown(send))
        Send(exampleClass, true);
}

protected override void Receive(Message message)
{
    var span = message.Buffer;
    Read(span, exampleClass, true);
}
```

> The `boolean` parameters for `SendJSON` and `ReadJSON` enable logging output by passing the contents of `span` and the resulting JSON representation to the `Log(string message, Object obj = null)` method defined by the included `Router` implementation.

Use the Inspector to populate one of the `ExampleClass` instances. Press the specified key and observe the members of the other instance being updated after the message is sent over the network.

Bonus: Schedule Messages

Add the following fields:

```
[SerializedField] KeyCode schedule;
[SerializedField] int interval;
Scheduler scheduler;
```

Add a `disposed` bool and override `OnDispose(bool disposing)` to dispose `scheduler`.

```
bool disposed;

protected override void OnDispose(bool disposing)
{
    base.OnDispose(disposing);

    if (disposing)
    {
        disposed = true;
        scheduler.Dispose();
    }
}
```

Modify `Disable()` to stop `scheduler`.

```
protected override void Disable()
{
    //...
    scheduler.Stop();
}
```

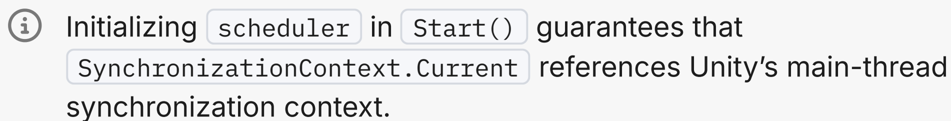
Add an `Execute()` method that sends a `DateTime` value if disposal has not occurred.

```
private void Execute()
{
    if (!disposed)
        Send(DateTime.UtcNow.Ticks);
}
```

`DateTime.UtcNow` ensures time-zone independent timekeeping.

Initialize `scheduler` in `Start()`.

```
private void Start()
{
    //...
    scheduler = new(SynchronizationContext.Current, Execute);
}
```



Initializing `scheduler` in `Start()` guarantees that `SynchronizationContext.Current` references Unity's main-thread synchronization context.

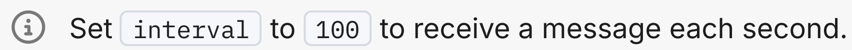
Modify `Update()` to toggle the scheduler if the `schedule` key was pressed down that frame.

```
protected void Update()
{
    //...

    if (Input.GetKeyDown(schedule))
    {
        if (scheduler.Paused)
            scheduler.Start(interval);
        else
            scheduler.Stop();
    }
}
```

Modify `Receive(Message message)` to log the received `DateTime.Seconds` to the console.

```
protected override void Receive(Message message)
{
    if (Read(message.Buffer, out DateTime dateTime, out int
count))
        Debug.Log($"received: {dateTime.Second} from:
{message.Route.EndPoint}");
}
```

Use the inspector to set the `schedule` and `interval` values, then start the play mode and press the `schedule` key down to start or stop the scheduler.



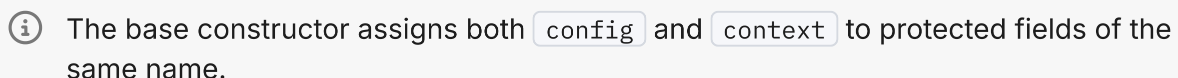
Set `interval` to `100` to receive a message each second.

Create a Latency Protocol

This tutorial demonstrates how to implement a custom protocol that simulates network latency by delaying inbound and outbound packets, using the `Packet`, `Header`, `Cache`, `Protocol`, and `Scheduler` types.

Create a `Latency` class that derives from the included `Protocol` implementation and define a `Scheduler` for each message path, with corresponding handler methods. Implement a constructor that calls the base constructor and initializes each scheduler using the provided `context` and the respective handler method.

```
public class Latency : Protocol
{
    public Latency(Configuration config, SynchronizationContext
context) : base(config, context)
    {
        inScheduler = new(context, In);
        outScheduler = new(context, Out);
    }

    readonly Scheduler inScheduler;
    readonly Scheduler outScheduler;

    private void In() {}
    private void Out() {}
}
```



> The base constructor assigns both `config` and `context` to protected fields of the same name.

For both the inbound and outbound paths, create a `Packet.Cache` instance, a packet ID counter, and a dictionary that maps each packet to the value returned by `DateTime.UtcNow` when the message was passed to the protocol.

```
int sendId;
int recvId;
readonly Packet.Cache inbound = new();
readonly Packet.Cache outbound = new();
readonly Dictionary<Packet.Cache.Key, DateTime> sends = new();
readonly Dictionary<Packet.Cache.Key, DateTime> recvs = new();
```

> ⓘ `sendId` and `recvId` are incremented for each packet to ensure that each `Packet.Id` is unique.

The included `Protocol` implementation defines a nested `Configuration` class to provide consumers with a means to configure public execution parameters.

Add the following fields to `Protocol.Configuration` :

```
public int LatencyIn = 100;
public int LatencyOut = 100;
```

Add the following properties to `Latency` :

```
DateTime Now => DateTime.UtcNow;
TimeSpan InboundDelay => new(0, 0, 0, 0, config.LatencyIn);
TimeSpan OutboundDelay => new(0, 0, 0, 0, config.LatencyOut);
```

> ⓘ Each `Protocol` maintains a reference to its `Configuration` object via its `config` field.

Override `OnOutbound(Message message)` and `OnInbound(Message message)` to handle messages passed to the protocol via its `Send(Message message)` and `Receive(Message message)` methods.

Convert each message into a packet by allocating a stack-based byte span large enough to hold both a header and the message contents. Increment the appropriate counter to generate a unique packet identifier, then store the packet in the corresponding cache. Record a timestamp for the packet, and start the scheduler for the path if it is currently paused.

```
 protected override void OnOutbound(Message message)
 {
     Span<byte> contents = stackalloc byte[Header.Size +
message.Buffer.Length];
     Header header = new(sendId++, default);
     Packet packet = new(header, message, contents);

     sends.TryAdd(packet.Key, Now);
     outbound.TryAdd(packet);

     if (outScheduler.Paused)
         outScheduler.Start(config.LatencyOut);
 }

protected override void OnInbound(Message message)
{
    Span<byte> contents = stackalloc byte[Header.Size +
message.Buffer.Length];
    Header header = new(recvId++, default);
    Packet packet = new(header, message, contents);

    recvs.TryAdd(packet.Key, Now);
    inbound.TryAdd(packet);

    if (inScheduler.Paused)
        inScheduler.Start(config.LatencyIn);
}
```

Override `Execute()` to call `In()` and `Out()`.

```
protected override void Execute()
{
    In();
    Out();
}
```

Within each of these methods:

Stop the respective scheduler for the path before processing to prevent overlapping execution.

Iterate through the appropriate packet cache in reverse order to safely remove entries while calculating the elapsed time since each packet was enqueued.

For each packet whose elapsed time meets or exceeds the configured delay, strip the protocol-added header to restore the original message, forward the resulting message via the appropriate callback (`InvokeInbound` or `InvokeOutbound`), and remove the packet from both the cache and tracking dictionaries.

If any packets remain pending, restart the scheduler with a dynamically adjusted delay based on the longest remaining elapsed time to ensure the next execution aligns with the earliest eligible packet, with respect to the configured delay.

```
private void In()
{
    if (!inScheduler.Paused) inScheduler.Stop();

    int interval = 0;

    for (int i = inbound.Count - 1; i >= 0; i--)
    {
        var packet = inbound[i];
        var key = packet.Key;

        if (recvs.TryGetValue(key, out var time))
        {
            var ts = Now.Subtract(time);

            if (ts >= InboundDelay)
            {
                Packet stripped = new(packet.Contents);
                InvokeInbound(stripped.Contents);
                inbound.Remove(key);
                recvs.Remove(key);
            }
            else interval = ts.Milliseconds > interval ?
ts.Milliseconds : interval;
        }
    }

    if (!inbound.IsEmpty)
        inScheduler.Start(config.LatencyIn - interval);
}
```

```
private void Out()
{
    if (!outScheduler.Paused) outScheduler.Stop();

    int interval = 0;

    for (int i = outbound.Count - 1; i >= 0; i--)
    {
        var packet = outbound[i];
        var key = packet.Key;

        if (sends.TryGetValue(key, out var time))
        {
            var ts = Now.Subtract(time);

            if (ts >= OutboundDelay)
            {
                Packet stripped = new(packet.Contents);
                InvokeOutbound(stripped.Contents);
                outbound.Remove(key);
                sends.Remove(key);
            }
            else interval = ts.Milliseconds > interval ?
ts.Milliseconds : interval;
        }
    }

    if (!outbound.IsEmpty)
        outScheduler.Start(config.LatencyOut - interval);
}
```

Override `OnClear(Route route)` to iterate through the inbound and outbound caches in reverse order and remove any packet whose route matches the `route` argument, along with its corresponding entry from either the `sends` or `recvs` dictionary. If either cache is empty after packets associated with the route have been cleared, stop the respective scheduler.

```
protected override void OnClear(Route route)
{
    for (int i = outbound.Count - 1; i >= 0; i--)
    {
        var key = outbound[i].Key;

        if (key.Route.Equals(route))
        {
            outbound.Remove(key);
            sends.Remove(key);
        }
    }

    if (outbound.IsEmpty)
        outScheduler.Stop();

    for (int i = inbound.Count - 1; i >= 0; i--)
    {
        var key = inbound[i].Key;

        if (key.Route.Equals(route))
        {
            inbound.Remove(key);
            recvs.Remove(key);
        }
    }

    if (inbound.IsEmpty)
        inScheduler.Stop();
}
```

Override `OnClear` to clear all collections and stop both schedulers.

```
protected override void OnClear()
{
    inbound.Clear();
    outbound.Clear();
    sends.Clear();
    recvs.Clear();
    inScheduler.Stop();
    outScheduler.Stop();
}
```

Override `OnDispose(bool disposing)` to call `base.OnDispose(disposing)`. If `disposing` is `true`, dispose both schedulers and both caches.

```
protected override void OnDispose(bool disposing)
{
    base.OnDispose(disposing);

    if (disposing)
    {
        inScheduler.Dispose();
        outScheduler.Dispose();
        inbound.Dispose();
        outbound.Dispose();
    }
}
```

The `disposing` argument indicates whether managed resources should be released.

Implement a Synchronous Node

This tutorial demonstrates how to implement a custom Node that executes synchronous dual-mode socket operations using the `NetSocketAddress`, `NetEndpoint`, and `NetSocket` types.

Create a new `ExampleNode` class derived from the included `Router` implementation and define the following members:

```
public class ExampleNode : Router
{
    [SerializeField] private int port;
    [SerializeField] private int bufferLength;
    [SerializeField] private bool dualMode;

    private NetSocket socket;
    public NetEndPoint LocalEndPoint { get; private set; }
}
```

Implement an `Awake()` method that initializes `LocalEndPoint` as a `NetEndPoint` created from the `port` serialized field and either a local IPv4 address or the IPv6 wildcard address. If `dualMode` is enabled, map the endpoint to IPv6 to support dual-stack operation. Initialize `socket` as a UDP `NetSocket` and bind it to `LocalEndPoint`.

```
private void Awake()
{
    var ipv4 = NetInfo.GetLocalAddress(AddressFamily.InterNetwork);
    var ipaddress = ipv4 ?? IPAddress.IPv6Any; //fallback to ipv6 in
the absence of ipv4
    var nsa = new NetSocketAddress(ipaddress, port);
    var sa = dualMode ? nsa.MapToIPv6() : nsa; //ensure socketaddr is
ipv6-mapped if dual-mode
    var ep = NetEndPoint.ThreadSafeCreate(sa);
    socket = new(sa.Family, SocketType.Dgram, ProtocolType.Udp);
    socket.DualMode = dualMode || ipv4 is null; //must set dual-mode
before binding
    socket.Bind(LocalEndPoint = ep);
}
```

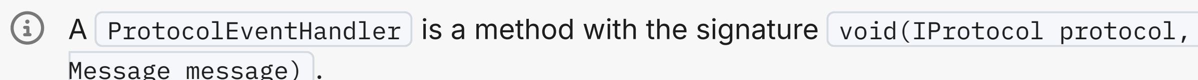
This allows `ExampleNode` to support both IPv4 and IPv6 on capable devices.

Implement a platform-independent `Error(NetEndPoint remoteEP, Span<byte> buffer, SocketError se)` method to be called if a send or receive operation fails.

```
private void Error(NetEndPoint remoteEP, Span<byte> buffer, SocketError
se)
{
#if UNITY_EDITOR_WIN || UNITY_STANDALONE_WIN
    Log($"{remoteEP} {se}", this);
#else
    Log($"{remoteEP} {se} errno: {(int)se}", this); //get the true
errno for Unix-like systems
#endif
}
```

Implement a `ProtocolEventHandler` that copies the message buffer, prepended by the route channel, into a stack-allocated buffer and sends it to `message.Route.EndPoint`, calling `Error(NetEndPoint remoteEP, Span<byte> buffer, SocketError se)` if the operation fails.

```
private void Send(IProtocol protocol, Message message)
{
    Span<byte> buffer = stackalloc byte[message.Buffer.Length +
sizeof(byte)];
    buffer[0] = message.Route.Channel;
    message.Buffer.CopyTo(buffer[1..]);

    var remoteEP = message.Route.EndPoint as NetEndPoint;

    if (!socket.SendTo(buffer, remoteEP, out SocketError error))
        Error(remoteEP, buffer, error);
}
```



A `ProtocolEventHandler` is a method with the signature `void(IProtocol protocol, Message message)`.

Override `Enable()` and `Disable()` to subscribe and unsubscribe the `ProtocolEventHandler` to the `Outbound` event of the underlying `Protocol` instance.

```
protected sealed override void Enable()
{
    base.Enable();
    Protocol.Outbound += Send;
}

protected sealed override void Disable()
{
    base.Disable();
    Protocol.Outbound -= Send;
}
```

Implement a corresponding method to reconstruct messages from received buffers by extracting a channel from the first index of each buffer to reconstruct the message route, leaving the remaining bytes as the message buffer, and constructing a new context using a timestamp.

Pass the resulting message to the underlying protocol to be received.

```
private void Receive(NetEndPoint remoteEP, Span<byte> buffer, DateTime
time)
{
    var channel = buffer[0];
    var context = new Context(time);
    var route = new Route(channel, remoteEP);
    Protocol.Receive(new Message<Context>(route, buffer[1..], ref
context));
}
```

Add a `Chronometer` field to track time; use `chronometer.Now` to get timestamps for message contexts.

Implement a `Process` method to poll the socket for available data, allocate a stack-based buffer sized to the `bufferLength` field, and receive buffers based on the socket's reported available data.

For each received packet, forward the received buffer and remote endpoint to the receive handler, with a timestamp for message reconstruction. If a receive operation fails, call the error handler with the associated endpoint, buffer, and error code.

```
private readonly Chronometer chronometer = new(DateTime.UtcNow);

void Process(NetSocket socket, NetEndPoint endpoint)
{
    var available = socket.Available;

    if (available > 0)
    {
        Span<byte> buffer = stackalloc byte[bufferLength];

        while (available > 0)
        {
            available -= bufferLength;

            EndPoint ep = endpoint;

            if (socket.RecvFrom(buffer, ref ep, out var error, out int
count))
                Receive(ep as NetEndPoint, buffer[..count],
chronometer.Now);
            else
                Error(endpoint, buffer, error);
        }
    }
}
```

Call `Process()` from a fixed-interval update loop to ensure consistent socket polling.

```
private void FixedUpdate()
{
    Process(socket, LocalEndPoint);
}
```

The remainder of this tutorial is dedicated to implementing dual-mode support.

In .NET, a `Socket` can send data only to destination addresses whose address family matches that of the `EndPoint` to which the socket is bound. Consequently, a .NET `Socket` bound to an IPv6 `EndPoint` must use an IPv6-mapped IPv4 address when sending to an IPv4 destination.

Add a `Dictionary` to maintain mappings between equivalent IPv4 and IPv6 endpoints, and implement a method that removes both endpoints when either one is provided.

```
private readonly Dictionary<NetEndPoint, NetEndPoint> endPoints =
new();

public bool Release(NetEndPoint endPoint)
{
    if (endPoints.TryGetValue(endPoint, out var remapped))
        endPoints.Remove(remapped);

    return endPoints.Remove(endPoint);
}
```

> ⓘ Using a `WeakDictionary` instead of a `Dictionary` would remove the need for explicit removal but risk premature garbage collection of mappings if references to the endpoints are not held elsewhere.

Modify the `ProtocolEventHandler` created earlier to check if `dualMode` is enabled and if `remoteEP` is IPv4. If both conditions are true, use a remapped instance to perform the operation.

Check `endPoints` for an equivalent IPv6-mapped instance. If none exists, create a `NetSocketAddress` from the original, remap it to IPv6, and add both the original and remapped instances to the dictionary so that each points to the other.

```
private void Send(IProtocol protocol, Message message)
{
    Span<byte> buffer = stackalloc byte[message.Buffer.Length +
sizeof(byte)];
    buffer[0] = message.Route.Channel;
    message.Buffer.CopyTo(buffer[1..]);

    var remoteEP = message.Route.EndPoint as NetEndPoint;

    if (dualMode && remoteEP.AddressFamily ==
AddressFamily.InterNetwork)
    {
        if (!endPoints.TryGetValue(remoteEP, out var remapped))
        {
            var sa = new
NetSocketAddress(remoteEP.SocketAddress).MapToIPv6();
            remapped = NetEndPoint.ThreadSafeCreate(sa);
            endPoints.Add(remapped, remoteEP);
            endPoints.Add(remoteEP, remapped);
        }

        remoteEP = remapped;
    }

    if (!socket.SendTo(buffer, remoteEP, out SocketError error))
        Error(remoteEP, buffer, error);
}
```

Modify the receive handler to check if the incoming `remoteEP` is an IPv6-mapped IPv4 address. If it is, use a remapped IPv4 instance for processing.

Check `endPoints` for an equivalent IPv4-mapped instance. If none exists, create a `NetSocketAddress` from the original endpoint, remap it to IPv4, and add both the original and remapped instances to the dictionary so that each points to the other.

```
private void Receive(NetEndPoint remoteEP, Span<byte> buffer, DateTime
time)
{
    if (remoteEP.SocketAddress.IsIPv4MappedToIPv6)
    {
        if (!endPoints.TryGetValue(remoteEP, out var remapped))
        {
            var sa = new
NetSocketAddress(remoteEP.SocketAddress).MapToIPv4();
            remapped = NetEndPoint.ThreadSafeCreate(sa);
            endPoints.Add(remapped, remoteEP);
            endPoints.Add(remoteEP, remapped);
        }

        remoteEP = remapped;
    }

    var channel = buffer[0];
    var context = new Context(time);
    var route = new Route(channel, remoteEP);
    Protocol.Receive(new Message<Context>(route, buffer[1..], ref
context));
}
```

Protocols interpret endpoints as distinct when their addresses use different formats (for example, IPv4 versus IPv6-mapped IPv4 for the same endpoint). To prevent such misidentification, use a consistent format for each address family.

Override `Clear()` to clear the `endPoints` collection.

```
protected override void Clear()
{
    base.Clear();
    endPoints.Clear();
}
```