Messages

So that's Events. Messages are an entirely different animal.

- Events is a Static class.
- Messages is a Service.
- Events is normally used as a store-and-forward events system: tiles post events and something else evaluates them later.
- Messages are sent immediately.

Messages can be sent to TilePlus (TPT) tiles; for example, to notify a tile of the player’s position. Messages can also be sent from one TPT tile (or from Event or Zone Actions) to other TPT tiles.

In response, a TPT tile can post an event to TpEvents. For example, after receiving a message about the player’s position and finding that it matches the player's position.

Rather than do something like "I want to control something on a tile on Tilemap ABC at position (3,2,0), what was that method I created last week?", use the TpMessaging Service Instead.

What's the advantage? Although you can use this Service to send messages to a tile on a Tilemap at a certain position, you usually use it to message several tiles on any tilemap, perhaps filtered for tile type, data packet contents, or a rectangular region.

You can send to all tiles, all with a particular tag, or all of a particular type or interface.

The TpMessaging Service uses generic interfaces and generic message packet classes. What does this mean?

`ServiceManager.MessagingService.SendMessage`

This has six different overloads, including one simple one which is rarely needed:

`SendMessage<T>(Tilemap map, Vector3Int position, T packet)`

Clearly this just corresponds to "I want to send a message...".

But let's talk about what this does and what T is in this context.

T is a TypeSpec for a concrete class inherited from an abstract class named `MessagePacket<T>`.

MessagePacket has two basic data items:

- SourceInstance: what sent the message

- Id: a uLong identifier for the message.

A useful simple packet is PositionPacket, which builds on MessagePacket to add a Vector3Int for position. This can be used to send a position of the player or an NPC to one or more TPT tiles.

Another simple packet is BoolPacket, which has (you guessed it) a boolean variable in it.

Similarly, StringPacket has a string.

For more complex use: ObjectPacket. This sort of like a 'Union' of several different types of fields, and is used extensively within the TilePlus system for events and Zone/Event Actions (discussed in other documentation).

AnimatorControlPacket is used by the TpZoneAnimator tile and an animator StateMachineBehaviour to allow a tile to control an Animator and get callbacks from the Animator.

ActionToTilePacket is used by the TpActionToTile Scriptable Object to send New Input System actions to tiles. See the section about 'ActionToTile'.

ObjectPacket, ActionToTilePacket, PositionPacket, and StringPacket all have pools accessible from the TpMessaging service. See various demos for how to use these. Please use the 'using' statement version of the pool access to ensure that the message packet instance is returned to the pool. It's up to you to ensure proper pooling use: if you can't, use unpooled message packet instances.

The Messaging Service maintains a context stack. The message packet and its Type are pushed on the top of the stack before messages are sent to tiles and popped off the stack after the messages are sent. The top of the stack can be 'peeked' so that message recipients can examine the original source packet. You can see how this is used in the Patterns demo program.

Messaging Methods

`SendMessage<T>(Tilemap map, Vector3Int position, T packet)`

is the simplest but least useful method.

If the TPT tile instance is available, you can use

`SendMessage<T>(TilePlusBase tile, T packet)`

Which is a convenience method for exactly the same thing since the tile has built-in properties to retrieve its parent Tilemap and its Grid position.

If you have the GUID of a TPT tile you can also use this:

`bool SendMessage<T>(Guid guid, T packet) where T:MessagePacket<T>`

- This is the only one of these methods which returns a value: false if the tile with a given GUID isn't found.
- C#'s GUID class has conversions between the string, byte, and Guid forms of a Guid. `Guid.TryParse()`
- This method requires the Guid form of a GUID.

If you know the map and have a list of positions:

```
SendMessage<T>(Tilemap map, IEnumerable<Vector3Int>   positions,
               T packet,
               Func<TilePlusBase, bool>? tileFilter = null)
```

Here we see the addition of a filter. This can be used to filter out tiles from those found. For example, you might want to examine some property of a tile and potentially exclude it from messaging by returning false from the filter callback.

TPT tiles can have tags. Here's how to message a group of TPT tiles with tags:

```
SendMessage<T>(Tilemap map, string tag,
             T                            packet,
             Func<TilePlusBase, bool>?    tileFilter       = null,
             Func<T, TilePlusBase, bool>? packetFilter = null,
             RectInt?                     region       = null)`
```

If `map` is null then every Tilemap with TPT tiles is examined to find tiles with tags.

Here we see the addition of a packetFilter. This filter is similar to the tileFilter with the addition of the packet itself as a filter parameter.

Notable is the addition of a region. If you supply a RectInt describing a region then that's used as an additional filter.

```
SendMessage<T>(Tilemap? map,
              Type                         typeSpec,
              T                            packet,
              Func<TilePlusBase, bool>?    tileFilter       = null,
              Func<T, TilePlusBase,bool>? packetFilter = null,
              RectInt?                  region = null)
```

Similarly, this overload sends the packet to all tiles of a particular Type, with both filtering varieties and the region filtering also available. Again, if `map` is null then every Tilemap with TPT tiles is examined to find tiles of matching Type.

```
SendMessage<T, TI>(T   packet,
                   Func<TI, TilePlusBase, bool>? interfaceAndTileFilter       = null,
                   Func<T, TilePlusBase,bool>?   packetFilter = null,
                   Func<TilePlusBase,bool>?   tileFilter = null,
                   RectInt?                      region       = null)
```

This is the most complex SendMessage overload. Here, there's no `map` specification at all, so all Tilemaps with TPT tiles are examined.

What's going on here? This can be used to send messages with packets of Type T to all tiles with a particular interface TI. There's an additional filter, the `interfaceAndTileFilter`. This filter gets the interface type, and the tile instance.

How Does a TPT Tile Handle Messages?

TPT tiles can handle multiple message packet Types by merely adding explicit interface declarations for whatever packets the tile wants to recieve. Here's a simple one from the Waypoint tile used in the Layout demo.

```
void ITpMessaging<PositionPacket>.MessageTarget(PositionPacket sentPacket)
{
    if (m_IsLevelChange)
    {
        //This tile won't respond until the game's Goal has been achieved. In this simple
game, get all of the chests.
        //It blinks to warn you and posts a message to the signboard.
        if (!gameState.SceneExitGoalAchieved)
            StartColorTween();
        else if (tweenerId != 0 && TweenerService)
            TweenerService.KillTween(tweenerId);
    }

    if (sentPacket.m_Position != m_TileGridPosition)
        return;
    ChangeSlide(true);
    isEnabled      = true;
    WasEncountered = true;
```

```
    TpEvents.PostTileEvent(this); //the WaypointEventAction plugin handles disabling other
WPs, saving data, level change etc.


}
```

and this is how the message is sent from within the LayoutDemoLayout Service (slightly edited for clarity):

```
using (messaging.PositionPacketPool.Get(out var pkt))
{
    pkt.m_Position     = newPlayerGridPosition;
    pkt.SourceInstance = null;

    messaging.SendMessage(null!, typeof(CdemoWaypointTile), pkt);

    //update all tiles that implement ITpMessaging<EmptyPacket,PositionPacket> with a new
position BUT filter out
    //the waypoints since they've been messaged already.
    //in this example, it's the NPC spawners, the ZoneSpawners etc
    if (!gameState.HadTileEvent) //don't send out this general message if an event had
occurred.
    {
        bool Filter(TilePlusBase tpb) => tpb.GetType() != typeof(CdemoWaypointTile);

        //This uses the simplest filter: the 'tileFilter'. Here we toss out all Waypoint
tiles.
        messaging.SendMessage<PositionPacket, ITpMessaging<PositionPacket>>(pkt, null, null,
Filter);
    }
}
```

Note that it's irrelevant what Tilemap the CDemoWaypointTiles are placed on. Since Waypoint tiles can change game state (such as changing to a new level) they're messaged first. The second SendMessage executes only if the first SendMessage call didn't result in any messaged tile posting an Event.

The second SendMessage call uses the more complex SendMessage call that allows you to only message tiles implementing ITpMessaging<PositionPacket>. The filter is the local method just above the second SendMessage call.

This must be terribly inefficient!!

You might think that one would need to tediously examine all these different Tilemaps to locate tiles to message, and it would be slow and inefficient. That would be true if SendMessage worked that way!

This isn't ye olde Unity GameObject `SendMessage`!!

Packets can only be sent to TilePlus tiles. These tiles 'register' themselves in TpLib data structures and 'deregister' themselves when they're deleted from a Tilemap.

The registration process stores Types, tags, interfaces, and other information that make locating tiles both fast and tilemap-independent. Tiles are found with fast Dictionary and HashSet lookups.

Here's how the overload with tags works:

```
public int[] SendMessage<T>(Tilemap map,
          string                       tag,
          T                            packet,
          Func<TilePlusBase, bool>?    tileFilter       = null,
          Func<T, TilePlusBase, bool>? packetFilter = null,
          RectInt?                     region       = null)
            where T : MessagePacket<T>
{
    SaveContext(packet);

    using (S_TilePlusBaseList_Pool.Get(out var list))
    {
        GetTilesWithTag(map, tag, list, tileFilter, region);

        var num = list.Count;

        if (packetFilter != null)
        {
            for (var i = 0; i < num; i++)
            {
                var tile = list[i];
                if (!messaged.Add(tile.Id))
                    continue;
                if (!packetFilter(packet, tile))
                    continue;
                var tgt = tile as ITpMessaging<T>;
                tgt?.MessageTarget(packet);
            }
```

```
        }
        else
        {

            for (var i = 0; i < num; i++)
            {
                var tile = list[i];
                if (!messaged.Add(tile.Id))
                    continue;
                var tgt = tile as ITpMessaging<T>;
                tgt?.MessageTarget(packet);
            }
        }
    }

    var ary = messaged.ToArray();
    PopContextAndDiscard();
    return ary;

}
```

The line `GetTilesWithTag(map, tag, list, tileFilter, region);` illustrates this. Without getting too deeply into the weeds here, that method examines an internal dictionary `s_TaggedTiles` to locate tiles to message.

It's very fast and (generally) scales linearly to larger number of tiles and/or tilemaps.

Revision #13
Created 2025-06-21 12:34:42 UTC by Vonchor
Updated 2025-09-16 12:20:00 UTC by Vonchor