Networking powers in Swift.

To offer depth and allow for interesting strategies, Swift needed to have powers that the player can choose and activate when they see fit.

My mission was to create the architecture for those powers, taking into account *networking* and *design iteration time*.

The Power class

We represented powers with Unity ScriptableObjects.

They are useful if you want to have objects with different settings but with the same behaviour. Designers can test multiple versions of the same power by duplicating the desired power asset.

The power base class contains virtual methods that all the different powers will override.

Power virtual methods

Powers have 3 distinct type of methods, signalled by their prefix.

Local : Will be called on the client that owns the instance of this power.

Remote : Will be called on the clients that do not own the instance of this power.

Server : Will be called on the server.

Power virtual methods

Here's the local code for the ShadowCloak power.

```
1 reference
private void ActivateCloakLocal()
{
    localSequence = DOTween.Sequence();

    localSequence.AppendCallback(() =>
    {
        localActive = true;

        fovBinding = player.character.AddFovModifier(fovModifier);
        canAttackBinding = player.attackHandler.canAttack.AddModifier(false);
        speedBinding = player.character.velocityMultiplier.AddModifier(speedBonus, ModificationType.Add);

        NetworkFXManager.ClientSpawnPP(pp, player.netIdentity);
    });

    localSequence.AppendInterval(lifespan);

    localSequence.AppendCallback(() =>
    {
        player.character.RemoveFovModifier(fovBinding);

        player.character.velocityMultiplier.RemoveModifier(speedBinding);

        NetworkFXManager.ClientDisablePP(pp, player.netIdentity);
    });

    localSequence.AppendInterval(reattackDelay);

    localSequence.OnComplete(() =>
    {
        player.attackHandler.canAttack.RemoveModifier(canAttackBinding);

        localActive = false;

        player.powerHandler.StartCooldown();
    });
}
```

Some important elements :

- Sequences

We use DOTween's Sequences to chain callbacks together.

- The active bool

The power is responsible for telling the PowerHandler when it starts and stops being active.

- StartCooldown()

The cooldown needs to be started by the power.

Power Handler

These methods will be called by the PowerHandler, a NetworkBehaviour that sits on the player GameObject. NetworkBehaviours are the only scripts that can call commands and receive RPCs in Mirror.

PowerHandlers have a serialized reference to all the powers in the game and will instantiate copies on Awake.

When a player wants to select a power, they will send a command to the server to request a different power. Until the server confirms the change, the player cannot use their power.

Power Handler

When the player tries to use their power, we need perform a few checks.

```
[Client]
1 reference
public void Activate()
{
    if (character.dead) return;
    if (clientIsChangingPower) return;
    if (CurrentPower == null) return;

    if (CurrentPowerCooldownNotOver || CurrentPower.localActive)
    {
        if(!player.headless) SoundManager.PlaySimple(GameUI.disabledActionSFX, transform.position);
        return;
    }
```

If these checks pass, we know we can call the activate method locally, but that doesn't mean the power will be able to start.

eg. A power that is cast on allies will have to abort if no ally is present.

Power Handler

This is why the LocalActivate method on the Power class returns a bool.

```
using (PooledNetworkWriter writer = NetworkWriterPool.GetWriter())
{
    if (instantiatedPowers[currentPowerIndex].LocalActivate(writer))
    {
        CmdActivate(writer.ToArray());

        onPowerLocalActivate?.Invoke(instantiatedPowers[currentPowerIndex]);
    }
}
```

The LocalActivate override on the power can return false at any time to cancel the activation.
In addition, a NetworkWriter is passed to the method to serialize any data that need to be sent to the server with a Command.

eg. If the power needs to have a player selected, we will send the id of the selected player.

Power Handler

Once the server receives the command, it will send the RPCs to the other players and pass the serialized data as well.

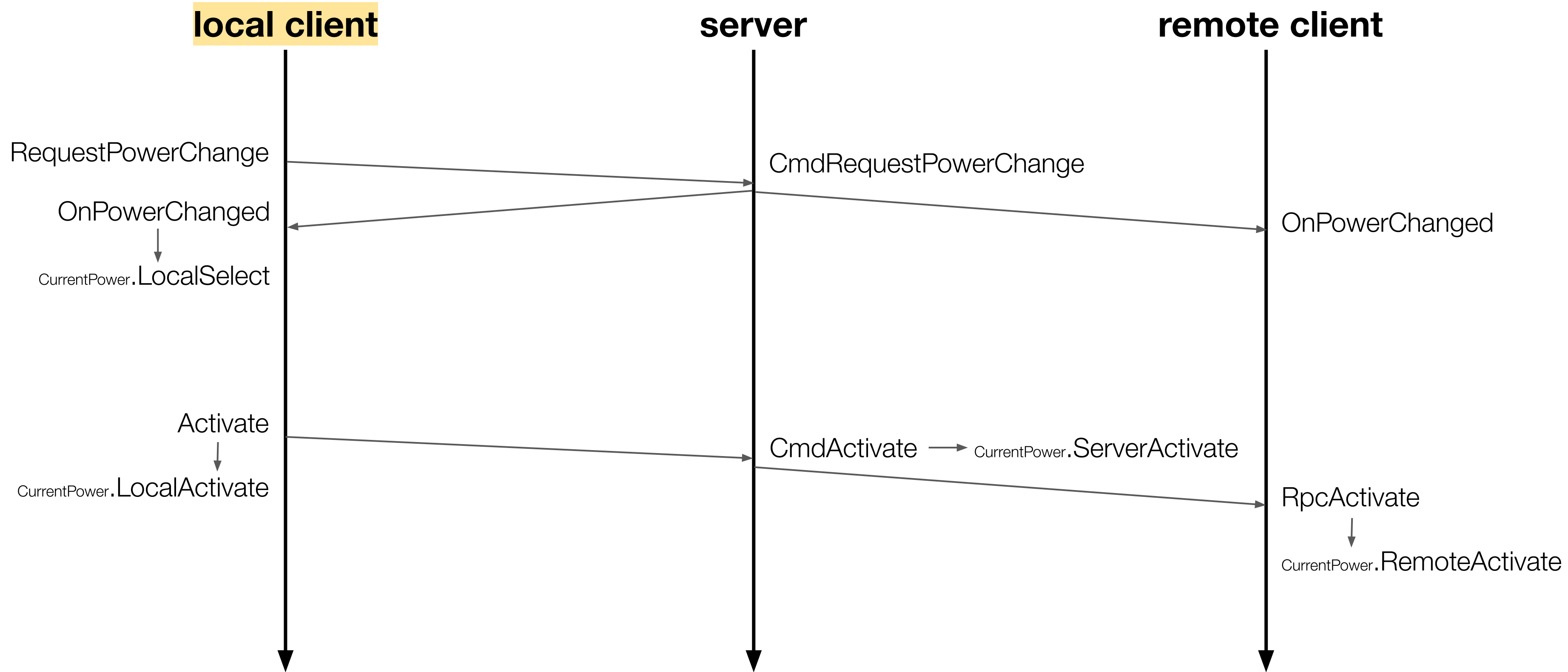
```
[Command]
1 reference
public void CmdActivate(byte[] bytes)
{
    instantiatedPowers[currentPowerIndex].ServerActivate(bytes);

    RpcActivate(currentPowerIndex, bytes);
}
[ClientRpc(excludeOwner = true)]
1 reference
public void RpcActivate(int powerIndex, byte[] bytes)
{
    instantiatedPowers[powerIndex].RemoteActivate(bytes);

    onPowerRemoteActivate?.Invoke(instantiatedPowers[powerIndex]);
}
```

As security is not a concern for a student game, we do not do any checks server-side. But we could also return a bool for ServerActivate and abort if cheating is detected.

Architecture



Cancelling powers

In addition to having Activate methods, powers have Stop methods. They need to be implemented for each Local, Remote and Server case.

Since we use DOTween Sequences for most of the powers, we use the Complete method to execute all Callbacks and OnComplete delegates instantly, arriving at the same result as if the power completed on its own.

```
8 references
public override void ServerStop()
{
    base.ServerStop();

    serverSequence.Complete();
}
```