.NET Conf China
2022

.NET低成本边缘计算方案

黄海鹏
苏州易泰勒电子科技有限公司软件研发总监

背景

Q：用一句话如何形容我的老板和客户的共同点？

A：他们都是老板？ 错

既要马儿跑得快，又要马儿不吃草。

背景

1. 终端节点部署持续增多，从单实例挂载门店变为混合多实例多门店
2. 服务器成本在增加，运维成本也在增加
3. 市场拓宽到海外，对本土研发和实施带来挑战
4. 越来越激烈的市场竞争，以及大的经济环境，对成本日趋敏感

1. 设备选型　　更低成本设备
2. 技术栈选型　　资源丰富，高效便捷的技术栈
3. 一些惯例做法　　降低服务器负载，将部分算力和带宽下沉
4. 安全和运维经验　低成本≠低质量

Why .NET?

我们团队使用.NET作为主要的技术栈，主要是因为：

1. 早期的工作经历背景，主要是偏工控行业的项目经验积累
2. .NET的持续迭代带来的优势，开源，跨平台.....
3. .NET一站式解决方案的能力，Web/App、服务端、客户端、嵌入式
4. 学习上手速度快，今年最新加入的四位同事都是没有接触过.NET的

.NET nanoFramework

.NET nanoFramework包含4个方面：

1. 对嵌入式主板的支持，包括ESP32、STM32、NXP、TI和M5Stack
2. 对主流设备的支持，包括传感器、显示屏等输入输出设备和传感器
3. 对通信协议的支持，包括MQTT、SignalR、WebSocket等等
4. 基础类库、示例代码等

ESP-WROOM-32 + BMP280

Running my .NET nanoFramework for 8 years on a battery

--Laurent Ellerbach

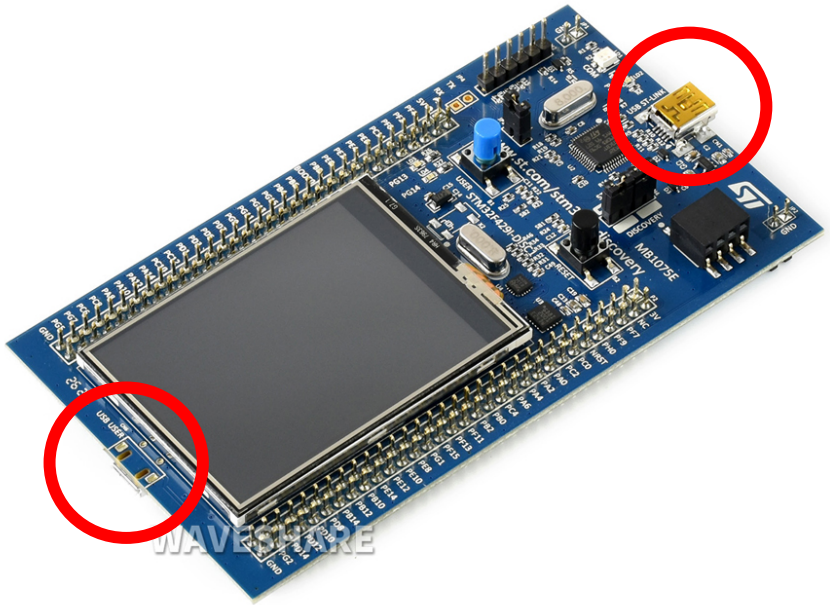
.NET nanoFramework vs .NET IoT

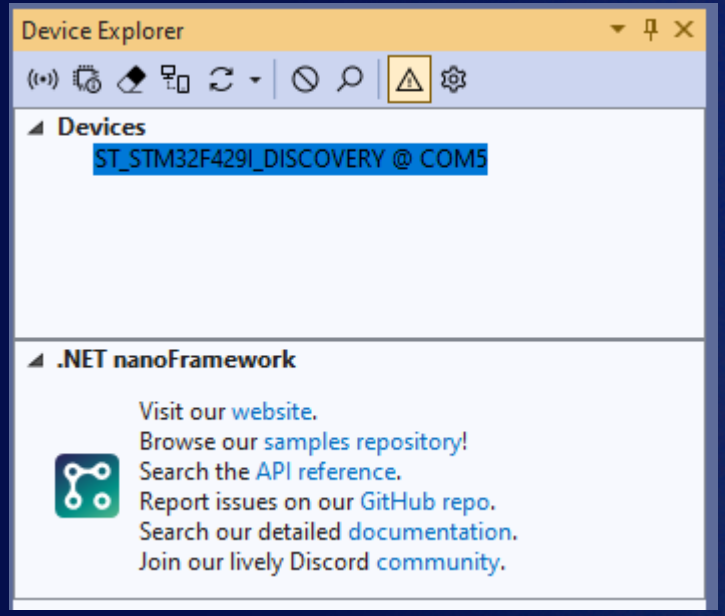
	.NET nanoFramework	.NET IoT
硬件资源	MCU，数MB内存	CPU，512MB或更多内存
.NET库	专用的nanoFramework库	都支持
设备关联	需要为特定的传感器引用特定的包	IoT.Device.Bindings和System.Device.Gpio
调试	需要根据经验推断	支持本地和远程调试
网络连接	自己写	操作系统（OS）支持
其他		
SSH	不支持/自己写	支持
证书	pem/crt/der	pfx

STM32F420I-DISC1

STM32

>> STM32 ST-LINK Utility (STSW-LINK004) :
https://www.st.com/en/development-tools/stsw-link004.html
注意有些版本需要同时连接mini-USB和micro-USB两个接口

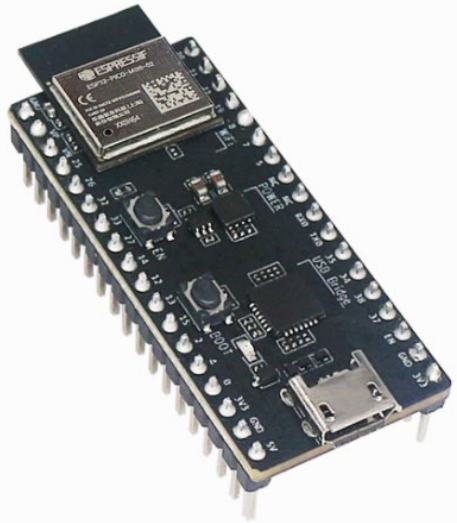


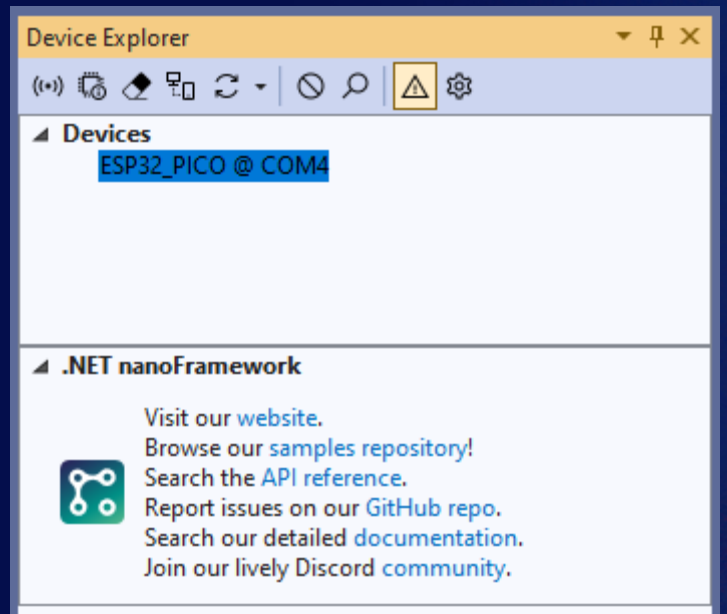


ESP32-PICO-MINI-02

ESP32

- ESP32设备，需要移步官网上下载ESP-IDF：
  - https://dl.espressif.com/dl/esp-idf-tools-setup-2.3.exe
- 对于VS Code，建议使用插件：
  - Espressif IDF，并且在安装后进一步安装ESP-IDF。
- 注意：如果出现Exit





XXX Pi

- 在价格面前，树莓派已经对国产派无话可说
- 国产派丰富多样，简直看花了眼，几乎是要啥有啥
- 技术支持方面，国产派只有一句话：自己研究（真人真事）
- 遇到问题不要慌，外部看门狗

惯例：主要思路

通信方面：

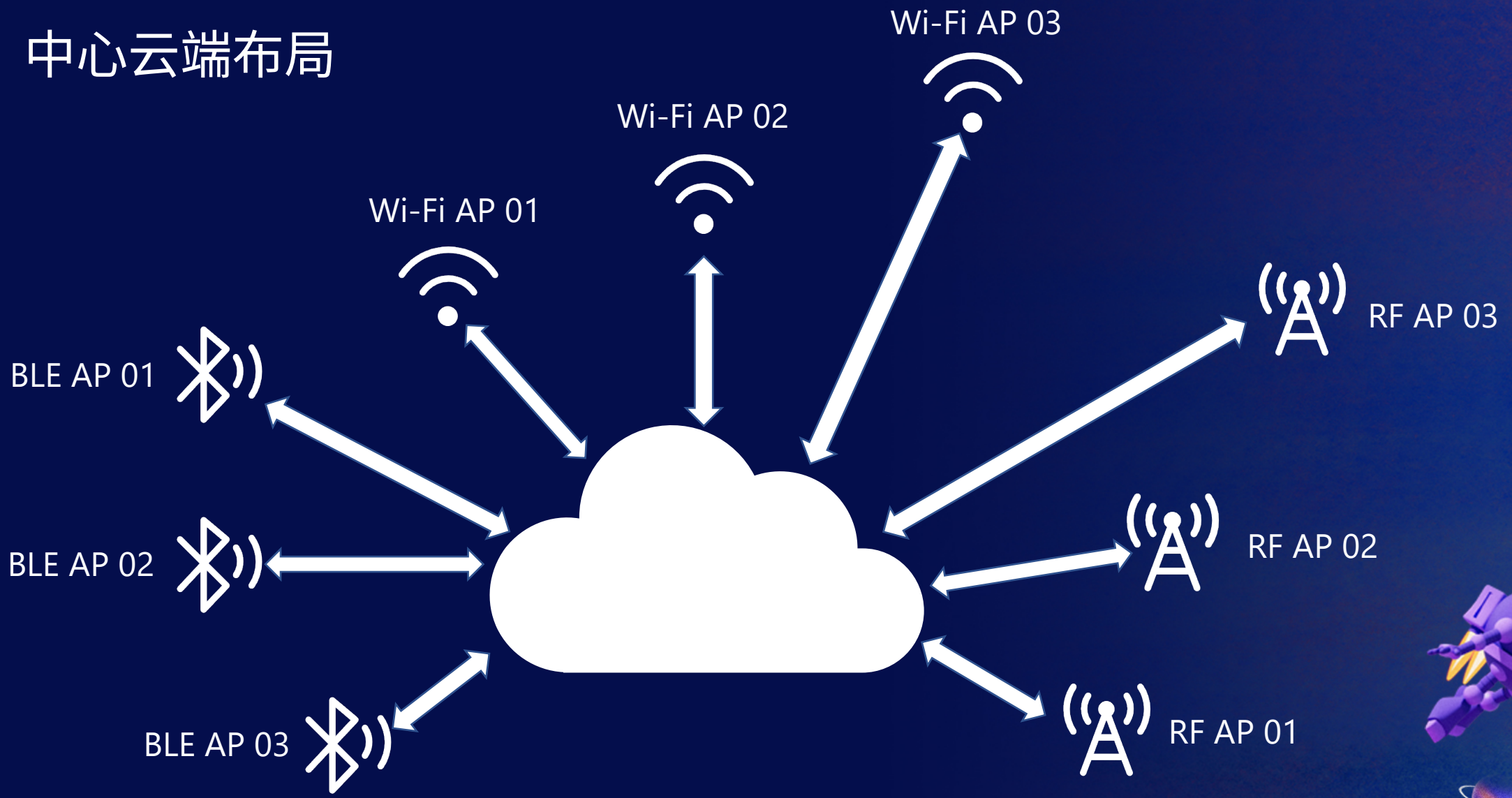
1. 降低连接数，但是避免唯一代理设备带来地短板
2. 降低收发次数，将数据合并后下发到现场或返回至服务端
3. 优化数据长度，采用更短小地数据格式，如MessagePackage

计算方面：

1. 基于数据包的通信，将服务端的一些计算下沉到边缘设备
2. 服务端专注于数据的持有，边缘设备专注于数据的处理

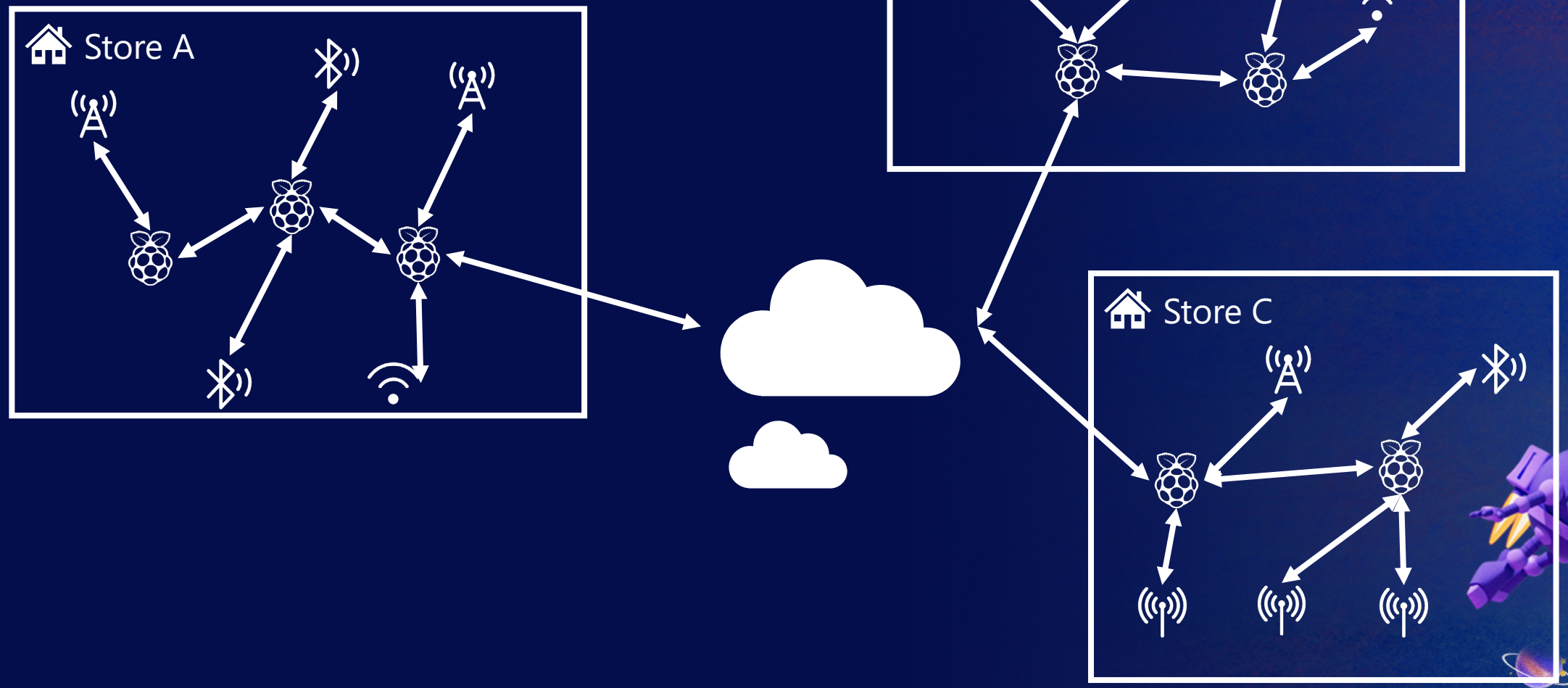
常见的项目布局（1）

中心云端布局



常见的项目布局（2）

多层级布局



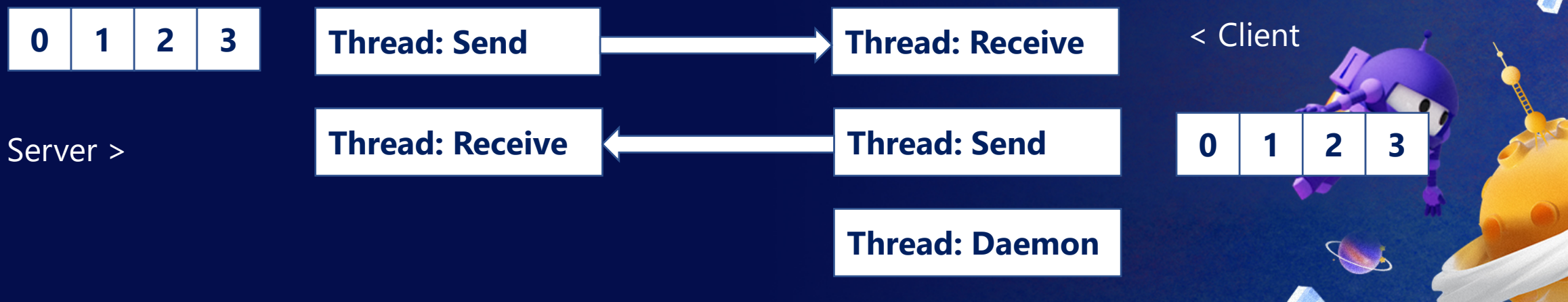
惯例（1）：收发机制

正常数据

- 待发数据存放在ConcurrentQueue中，发送Peek，确认Dequeue
- 双向确认：采用Token标记，Byte/WORD/GUID

心跳数据

- 无需确认
- 于收发线程之外的连接保持线程
- 除了单纯的心跳含义，还需要考虑设备的状态参数



惯例（1）：收发机制 Keep


```
Task.Factory.StartNew(async () =>  // Thread to keep connect
{
  while (true)
  {
    try
    {
      if (_socket.State != WebSocketState.Open && _socket.State != WebSocketState.Connecting)
      {
        if (Status != ClientStatus.Disconnect) { /* Disconnect Logic */ }
        await _socket.ConnectAsync(_uri, _cancel);
        await _socket.SendAsync(registerData, WebSocketMessageType.Binary, true, _cancel);
        await Task.Delay(TimeSpan.FromSeconds(15));
        continue;
      }
      else if (_socket.State == WebSocketState.Open)
      {
        if ( /* Heartbeat Logic */)
        {
           await _socket.SendAsync(heartbeat, WebSocketMessageType.Binary, true, _cancel);
        } // Dealy and continue here…
      } // Other logic here…,
    }
    catch (Exception ex) { /* Exception handler logic */ }
});
```

惯例（1）：收发机制 Recieve


```
Task.Factory.StartNew(async () =>  // Thread to keep sending
{
  List<ArraySegment<byte>> cache = new();
  while (true)
  {
    try
    {
      if (_socket is null || _socket.State != WebSocketState.Open) { /* Wait logic */}
      byte[] buffer = new byte[1024 * 8];
      try
      {
         var result = await _socket.ReceiveAsync(new ArraySegment<byte>(buffer), _cancel);
         if (result.Count > 0)
         {
           var temp = new byte[result.Count];
           Array.Copy(buffer, temp, result.Count);
           cache.Add(temp);
         }
         if (!result.EndOfMessage) continue;
      }
      catch (Exception e0) { /* Delay logic */}
      var data = new byte[cache.Sum(x => x.Count)]; if (data.Length == 0) continue;
      var index = 0; var dummyData = MessagePackSerializer.Deserialize<DummyData>(data);
    } catch (Exception ex) { /* Exception handler logic */ } }});
```

惯例（1）：收发机制 Recieve

```
    // 承接上段
    var data = new byte[cache.Sum(x => x.Count)];
    var index = 0;
    foreach (var item in cache)
    {
      Array.Copy(item.ToArray(), 0, data, index, item.Count);
      index += item.Count;
    }

    if (data.Length == 0) continue;
    var finalData = MessagePackSerializer.Deserialize<YOUR_DATA_TYPE>(data);
    cache.Clear();  // DO NOT MOVE TO FINAL
    catch (Exception ex)
    {
      // Exception handler logic
      cache.Clear(); // DO NOT MOVE TO FINAL
    }
  }
});
```

惯例（1）：收发机制 Send


```
Task.Factory.StartNew(async () =>  // Thread to keep sending
{
  while (true)
  {
    try
    {
      if (_socket is null || _socket.State != WebSocketState.Open || DataBuffer.IsEmpty)
      {
        await Task.Delay(TimeSpan.FromSeconds(2)); continue;
      } // 连接状态检测，待发数据队列

      if (DataBuffer.TryPeek(out var data) && data.Count > 0)
      {
          await _socket.SendAsync(data, WebSocketMessageType.Binary, true, _cancel);
          DataBuffer.TryDequeue(out data);
      }
    }
    catch (Exception ex) { /* Exception handler logic */ }
   }
});
```

惯例（2）：差速缓冲机制

差速缓冲机制用于解决低功耗设备和边缘计算设备、边缘计算设备和服务端之间异步通信和网络波动问题。一般认为，在多个边缘设备的现场网络中，网络的环境和质量，是比较稳定的。

不同于RPC/gRPC的思路，服务端不必关心边缘设变如何与低功耗设备之间的通信细节，只将任务数据推送给边缘设备。服务端专注于数据的持有，边缘设备专注于任务的处理。

1. 应发尽发：服务端尽可能地将数据发送给边缘计算
2. 专心工作：边缘计算专注于场内地通信与数据处理
3. 耐心等待：双方保持对一个连接的倒计时，超时后尝试或切换

惯例（2）：差速缓冲机制


```
readonly ConcurrentQueue<string> _cache = new();
private void SerialPort_DataReceived(object sender, SerialDataReceivedEventArgs e)
{
  try {_cache.Enqueue(_serialPort.ReadExisting());}
  catch (Exception ex) { /* Error Handler */}
}
private async Task DataRead()
{
  while (true)
  {
    try
      {
        if (_cache.IsEmpty) { /* Delay logic */ }
        while (_cache.TryDequeue(out var data))
        {
          if (string.IsNullOrEmpty(data)) continue; // No data
          _result.Append(data);
        }

        if (_result.Length == 0) { /* Delay logic */ }
        while (_result.Length > 0 && _result[0] != '*') _result.Remove(0, 1); // Filter Logic
        // Some other logic here, may parse data and convert to your data object…
        RecieveDataHandler?.Invoke(Port, code, GetResults(result.Substring(14, length)));
        _result.Clear();
```

惯例（2）：差速缓冲机制

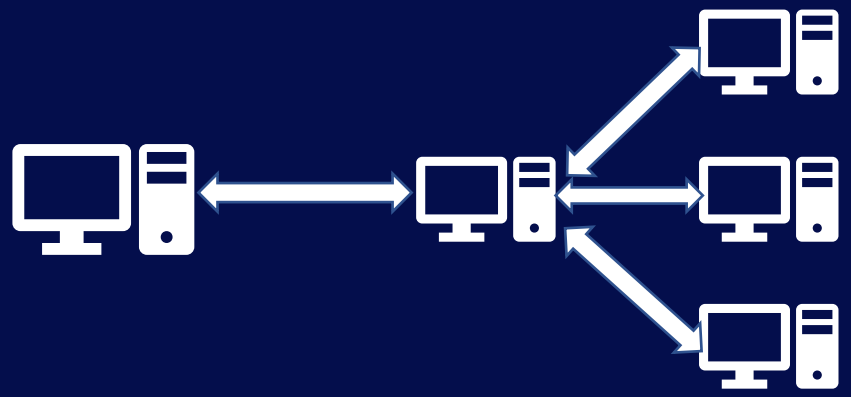

```
serailPort.RecieveDataHandler += ProcessResult;

static void ProcessResult(int port, Result code, List<TagResult>? result)
{
  // Some logica here: convert code&result =>report
  Dummy.Push(MessagePackSerializer.Serialize(report));
  Proxy.Broadcast(clear.ToArray());
}

public void Push(byte[] message, DataCode code, Guid token, bool immediately = false)
{
  try
  {
    var data = new Data { Code = code, Data = message, Token = token };
    var array = new List<byte> { DataPrefix.Object };
    array.AddRange(MessagePackSerializer.Serialize(data));
    if (immediately) {
       if (_socket.State == WebSocketState.Open)
         _socket.SendAsync(message, WebSocketMessageType.Binary, true, _cancel).Wait();
    } else {
       DataBuffer.Enqueue(array.ToArray()); }
    }
    catch (Exception ex) { /* Error Handler */ }
 }
```

惯例（3）：转发机制

- 降低TCP连接数，同一局域网仅维持一个连接
- 数据包增加最终接受设备地址
- 代理设备进行数据合并
- 由服务端裁定连接（简单），局域网内选举（复杂）



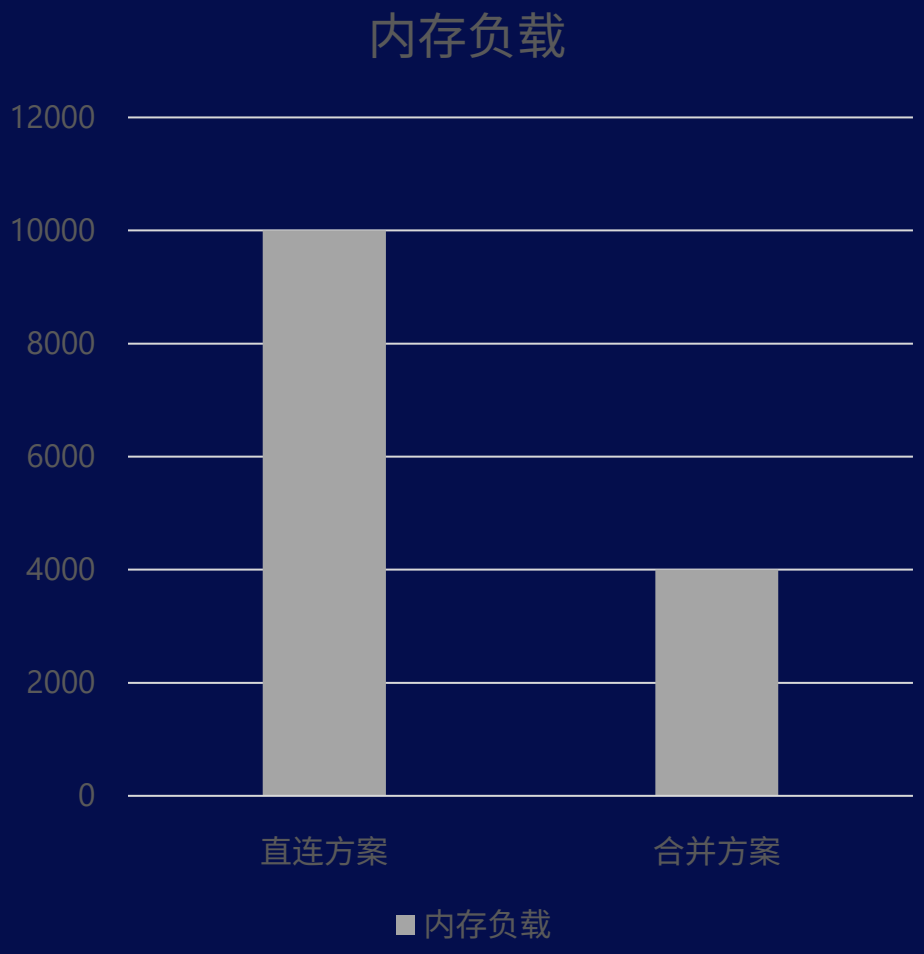
Message List
Device #1
Message AAA
Device #2
Message 123
Device #1
Message @#$

Device #1
Message AAA

Device #2
Message 123

Device #1
Message @#$

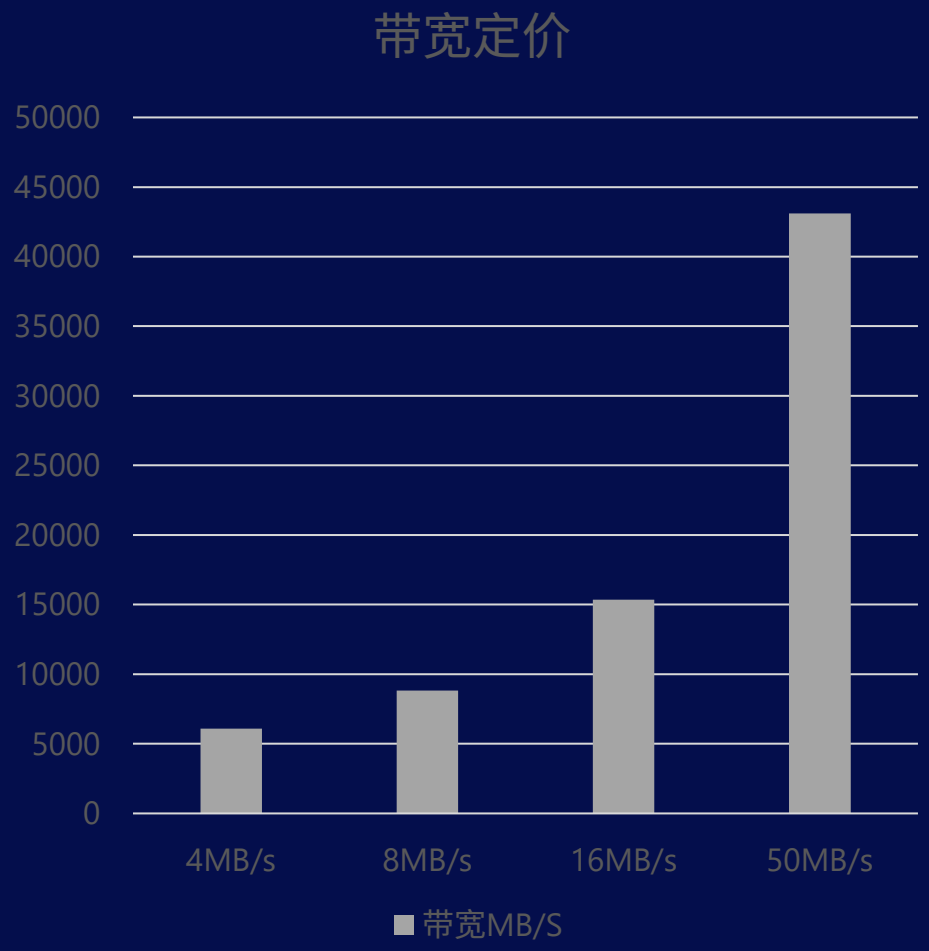
惯例（3）：转发机制



假定某连锁超市客户门店数量在500家，
平均每个门店部署的基站在10台。
假定我们原先为每个连接分配的
BufferSize是2MB，现在提高到8MB。

阿里云一台标准ESC服务器，
32G内存要比16G内存高**50%**的报价。
Azure一台虚拟机，
56G内存要比14G内存高**83%**的报价。

惯例（3）：转发机制



假定某连锁超市客户门店数量在300家，平均每个门店部署的WiFi连接数是200个。假定每台设备更新的视频和图像内容在20MB，以每天80%的更新量发生在峰值更新4个小时计算，流量大约在50MB/秒。如果其中有80%的视频是重复的，流量大约在13.5MB/秒

50MB的带宽报价大概是16MB的带宽报价的**2.8**倍。

惯例（4）：副本机制

数据与容器化思路相似，即：
设备不持久化存储数据，只保留临时数据，和短期的日志

副本机制除了数据初始化外，还需要固件初始化，即：
在无法启动当前版本固件的时候，从最开始初始化固件启动

对设备和应用程序存在一定的宽容度，即：
允许出现芭比Q的情况，只要能够自我恢复即可。

安全与可控

Q: OEM可能存在非授权/计划外的生产
A: 预分配法：在设备生产时，确认设备身份：ID/MAC
Q: 设备身份与通信数据的伪造可能
A: 三元组法：ProductKey + DeviceName + DeviceSecret
Q: 网络或服务端异常时的应急措施
A: 应急通信：PDA/APP预存，通过蓝牙与设备连接
Q: 设备缺陷、系统缺陷或者应用缺陷
A: 软件看门狗+硬件看门狗方案

生产与售后

自动化测试：在灌录的镜像首次开机的时候，增加自动化测试的内容。如依次检测网络、接口、显式、通信各个模块的内容。降低OEM人力成本。

快捷售后：当某个设备处于故障状态时，降低客户/客服的售后成本，对替换设备采用开箱即用的方式。在云端或者外部插拔设备自动配置设备参数。

OTA：灰度OTA、自动化OTA


.NET Conf China
2022
开源 · 安全 · 赋能
THANK YOU!
感谢聆听