

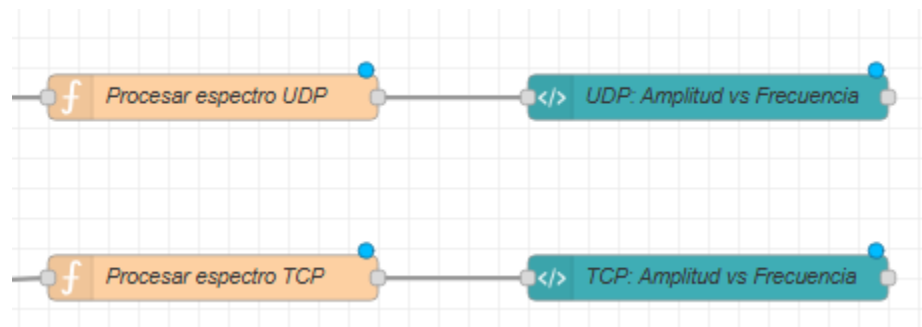
Apéndice G. Procesamiento de Datos en Node-RED

1. Acumulación del Búfer

En Node-RED, dentro de un nodo Function dedicado a esta tarea, cada vez que llega un fragmento de datos (ya sea por UDP o por TCP) se guarda en un espacio temporal llamado “buffer parcial”. El nodo no envía nada a la siguiente etapa mientras el recipiente no alcance el tamaño exacto de una ventana espectral completa (por ejemplo, N igual 32 o 1024 muestras).

Figura 1

Procesar espectro UDP/TCP.



Nota. La imagen muestra los nodos responsables del procesamiento de los datos ingresados a Node-red desde GNU Radio.

El proceso es así:

Recepción y unión, cada trozo de datos que entra se concatena al contenido que ya había quedado guardado. De este modo, aunque lleguen paquetes muy pequeños o desordenados, siempre se van acumulando en el mismo espacio de forma continua.

Verificación de integridad, antes de intentar procesar, el nodo revisa que el total de bytes en el buffer sea múltiplo de 4, es decir, que corresponda a un número entero de valores ‘float’

(cada valor ocupa 4 bytes). Si no lo es, significa que falta información para completar el próximo número en la secuencia, así que el nodo detiene su marcha, conserva lo acumulado y vuelve a esperar más datos.

Envío de la ventana completa, cuando el buffer alcanza el tamaño justo de una ventana espectral (o incluso supera ese tamaño), el nodo extrae los datos necesarios para formar esa ventana reuniendo las muestras y los metadatos de frecuencia y lo envía de una sola vez hacia el siguiente nodo.

Guardado del sobrante, si tras extraer la ventana completa queda información adicional en el buffer (por ejemplo, parte de la siguiente ventana), ese remanente se guarda para la próxima iteración. Así, nada se pierde ni se confunde entre ventanas sucesivas.

2. Código del nodo Procesar espectro UDP

Figura 2

Código procesar espectro UDP parte A.

```

1  // -----
2  // Nodo Function: Procesa UDP con Acumulación y salida unificada (usa Flow Context)
3  // -----
4
5  // Recuperar el buffer parcial acumulado del flujo (flow context)
6  let partialBuffer = flow.get('partialBuffer') || Buffer.alloc(0);
7
8  // Concatenar el nuevo mensaje al buffer acumulado
9  partialBuffer = Buffer.concat([partialBuffer, msg.payload]);
10 const totalBytes = partialBuffer.length;
11
12 // Si totalBytes no es múltiplo de 4, esperar a que lleguen más datos.
13 if (totalBytes % 4 !== 0) {
14   flow.set('partialBuffer', partialBuffer);
15   return null;
16 }
17
18 // Convertir el buffer acumulado a un arreglo de floats.
19 const floatCount = totalBytes / 4;
20 const data = [];
21 for (let i = 0; i < floatCount; i++) {
22   data.push(partialBuffer.readFloatLE(i * 4));
23 }
24
25 // Función auxiliar para hallar índices exactos de un valor.
26 function findExactIndices(arr, value) {
27   return arr.map((v, i) => Object.is(v, value) ? i : -1)
28     .filter(i => i !== -1);
29 }
30
31 const startIndices = findExactIndices(data, -1.0);
32 const endIndices = findExactIndices(data, -2.0);
33

```

Nota. La imagen muestra el código parte A responsable del procesamiento de los datos.

Figura 3

Código procesar espectro UDP parte B.

```

33
34 const plotlyMsgs = [];
35 let processedBytes = 0;
36
37 // Procesar al menos un paquete completo (si lo hay)
38 if (startIndices.length > 0) {
39   for (let s = 0; s < startIndices.length; s++) {
40     let startIdx = startIndices[s];
41     let validPacket = false;
42     for (let i = 0; i < endIndices.length; i++) {
43       let endIdx = endIndices[i];
44       if (endIdx > startIdx) {
45         const totalElements = endIdx - startIdx + 1;
46         if (totalElements >= 6) {
47           const N_dynamic = totalElements - 6;
48           if (!Object.is(data[endIdx], -2.0)) {
49             node.warn(`❌ Paquete UDP en índice ${startIdx} ignorado: bandera final incorrecta.`);
50             continue;
51           }
52           const f_inicial = data[startIdx + 1];
53           const f_paso = data[startIdx + 2];
54           const f_centro = data[startIdx + 3];
55           const espectro = data.slice(startIdx + 4, startIdx + 4 + N_dynamic);
56           const f_final = data[startIdx + 4 + N_dynamic];
57           const ejeX = espectro.map((_, i) => f_inicial + i * f_paso);
58
59           // Preparar el mensaje para Plotly
60           plotlyMsgs.push({
61             payload: { x: ejeX, y: espectro }
62           });
63
64           node.warn(`✅ Paquete UDP válido desde índice ${startIdx}: N=${espectro.length} muestras.`);
65

```

Nota. La imagen muestra el código parte B responsable del procesamiento de los datos.

Figura 4

Código procesar espectro UDP.

```

65 |
66 |                                     processedBytes = (endIdx + 1) * 4;
67 |                                     validPacket = true;
68 |                                     break;
69 |                                 }
70 |                             }
71 |                         }
72 |                     if (validPacket) break;
73 |                 }
74 |             }
75 |
76 | // Eliminar la parte procesada del buffer.
77 | if (processedBytes > 0) {
78 |     const remainingBuffer = partialBuffer.slice(processedBytes);
79 |     flow.set('partialBuffer', remainingBuffer); // Se asegura de guardar el buffer restante
80 |     if (plotlyMsgs.length > 0) {
81 |         return plotlyMsgs[0];
82 |     } else {
83 |         return null;
84 |     }
85 | } else {
86 |     flow.set('partialBuffer', partialBuffer); // Guardar el buffer parcialmente procesado
87 |     return null;
88 | }
89 |

```

Nota. La imagen muestra el código responsable del procesamiento de los datos ingresados a Node-RED por el protocolo UDP.

2.1. Procesar Espectro UDP

Almacenamiento en flow context, el nodo recupera del “flow context” el buffer parcial (partialBuffer) y añade el nuevo fragmento UDP recibido.

Conversión a números, una vez que hay suficientes bytes (múltiplo de 4), traduce todo ese bloque en un arreglo de números de punto flotante.

Detección de banderas, busca en ese arreglo los valores que marcan el inicio (-1.0) y el final (-2.0) de cada paquete.

Selección de un paquete válido, recorre los posibles pares de índices de inicio y fin. Solo considera un paquete si entre esas banderas hay al menos seis elementos (metadatos y muestras), y verifica de nuevo que la bandera de fin esté en la posición correcta.

Construcción de la salida, extrae los parámetros de frecuencia inicial y de paso, calcula el eje X correspondiente y reúne las muestras del espectro en un objeto listo para Plotly.

Limpieza y almacenamiento del resto, elimina del buffer los bytes ya procesados y guarda el sobrante para la siguiente vuelta. Solo entonces devuelve el mensaje con el espectro completo.

3. Código del nodo Procesar espectro TCP

Figura 5

Código procesar espectro TCP parte A.

```

1 // -----
2 // Nodo Function: Procesa TCP con acumulación y salida unificada (usa Flow Context)
3 // -----
4
5 // Recuperar el buffer parcial acumulado de TCP del flujo (flow context)
6 let tcpBuffer = flow.get('tcpBuffer') || Buffer.alloc(0);
7
8 // Concatenar el nuevo mensaje TCP al buffer acumulado
9 tcpBuffer = Buffer.concat([tcpBuffer, msg.payload]);
10 const totalBytes = tcpBuffer.length;
11
12 // Si totalBytes no es múltiplo de 4, es que no tenemos un float completo, esperar a que lleguen más datos.
13 if (totalBytes % 4 !== 0) {
14   flow.set('tcpBuffer', tcpBuffer);
15   return null; // Espera más datos
16 }
17
18 // Convertir el buffer acumulado a un arreglo de floats.
19 const floatCount = totalBytes / 4;
20 const data = [];
21 for (let i = 0; i < floatCount; i++) {
22   data.push(tcpBuffer.readFloatLE(i * 4));
23 }
24
25 // Función auxiliar para hallar índices exactos de un valor.
26 function findExactIndices(arr, value) {
27   return arr.map((v, i) => Object.is(v, value) ? i : -1)
28     .filter(i => i !== -1);
29 }
30
31 const startIndices = findExactIndices(data, -1.0);
32 const endIndices = findExactIndices(data, -2.0);
33

```

Nota. La imagen muestra el código parte A responsable del procesamiento de los datos.

Figura 6

Código procesar espectro TCP parte B.

```

33
34 const plotlyMsgs = [];
35 let processedBytes = 0;
36
37 // Procesar al menos un paquete completo (si lo hay)
38 if (startIndices.length > 0) {
39   for (let s = 0; s < startIndices.length; s++) {
40     let startIdx = startIndices[s];
41     let validPacket = false;
42     for (let i = 0; i < endIndices.length; i++) {
43       let endIdx = endIndices[i];
44       if (endIdx > startIdx) {
45         const totalElements = endIdx - startIdx + 1;
46         // El paquete debe tener al menos 6 elementos:
47         // [bandera_inicio, f_inicial, f_paso, f_centro, f_final, bandera_fin]
48         if (totalElements >= 6) {
49           const N_dynamic = totalElements - 6;
50           // Confirmamos que en la última posición esté la bandera de fin (-2.0)
51           if (!Object.is(data[endIdx], -2.0)) {
52             node.warn('❌ Paquete TCP en índice ${startIdx} ignorado: bandera final incorrecta.');

```

Nota. La imagen muestra el código parte B responsable del procesamiento de los datos.

Figura 7

Código procesar espectro TCP.

```

65
66     plotlyMsgs.push({
67       payload: { x: ejeX, y: espectro }
68     });
69
70     node.warn('✅ Paquete TCP válido desde índice ${startIdx}: N=${espectro.length} muestras.');

```

Nota. La imagen muestra el código responsable del procesamiento de los datos ingresados a Node-RED por el protocolo TCP.

3.1 Procesar espectro TCP

El funcionamiento es casi idéntico al caso UDP, con la única diferencia de que usa una variable de buffer separada (tcpBuffer) en el flow context.

Se concatena el nuevo fragmento TCP al buffer anterior.

Se verifica que el total de bytes permita una conversión a floats.

Se convierte todo el buffer a un arreglo de números de punto flotante.

Se localizan las banderas de inicio y fin dentro de ese arreglo.

Se valida que el paquete tenga al menos seis elementos y que la bandera de fin sea la correcta.

Se extraen los datos de frecuencia y las muestras del espectro, se arma el mensaje para Plotly, y se marcan los bytes procesados.

Finalmente, se guarda el resto del buffer para la siguiente iteración y el nodo envía el primer paquete completo al flujo de visualización.

4. Localización de banderas

Cuando el nodo Function ha reunido suficientes datos en el búfer y los convierte a una lista de números, llega el momento de “marcar” dónde empieza y dónde termina la porción útil de cada espectro. Para eso se utilizan dos valores especiales, conocidos como banderas:

Bandera de inicio (−1.0): señala el punto exacto en el que arranca la información de interés.

Bandera de fin (−2.0): indica dónde finaliza ese bloque de datos.

Piensa en estas banderas como postes que delimitan un campo de fútbol. Todo lo que esté entre el poste de inicio y el poste de fin constituye la ventana espectral que queremos procesar.

Node-RED las interpreta de la siguiente manera:

Búsqueda: el nodo revisa cada elemento del arreglo de números y anota en qué posiciones aparecen los valores -1.0 y -2.0 .

Emparejamiento: por cada bandera de inicio detectada, el nodo busca la bandera de fin que le sigue. Así, forma parejas coherentes (inicio–fin), asegurándose de no cruzar los límites entre un paquete y otro.

Validación mínima: solo considera un paquete válido si entre el de inicio y el de fin hay, al menos, la estructura mínima: los metadatos (frecuencia inicial, paso y centro) y las muestras reales. Si encuentra datos desordenados o insuficientes, descarta ese fragmento y continúa buscando.

Extracción: una vez que confirma el inicio y el fin de un bloque útil, copia precisamente todo lo que está en medio ni un número antes, ni un número después y lo prepara para el siguiente paso.

De esta forma, Node-RED utiliza esas banderas para recortar los datos en fragmentos exactos y sin ambigüedad, eliminando secciones sobrantes. Gracias a ese sistema, cada ventana espectral que llega a la visualización proviene de un paquete perfectamente delimitado.

Además de las banderas que encierran la ventana espectral (-1.0 , -2.0), el nodo Function también reconoce otro par de marcadores, -3.0 y -4.0 , que delimitan un bloque de constantes asociadas a cada paquete. Estas constantes incluyen la posición latitud y longitud, parámetros de orientación (salto, azimut y elevación), la altitud y una descripción codificada en caracteres.

Detección del bloque de constantes

Una vez que el nodo ha encontrado y extraído el espectro entre -1.0 y -2.0 , busca inmediatamente

la siguiente marca -3.0 , que señala el inicio del bloque de constantes, y luego el -4.0 , que indica su fin.

Emparejamiento secuencial, el nodo garantiza que estas banderas -3.0 y -4.0 aparezcan justo después de la bandera de fin espectral (-2.0). De este modo, cada paquete siempre trae primero su porción de datos de amplitud/frecuencia y, acto seguido, su conjunto de metadatos.

Extracción de valores, tan pronto confirma la posición de -3.0 y -4.0 , el nodo toma todos los números intermedios y los asigna, en orden, Latitud, Longitud, Salto (paso entre mediciones), Azimut, Elevación, Altitud, Descripción (interpretada convirtiendo cada código numérico en su carácter correspondiente).

Construcción del mensaje unificado, con la ventana espectral y el bloque de constantes completos, el nodo arma un único objeto `msg.payload` que incluye las 11 propiedades que espera la plantilla de la interfaz de usuario: las coordenadas, la hora local, los parámetros de frecuencia inicial y de paso, el número de muestras y la lista de datos, junto con los metadatos de posición y descripción.

Reinicio y continuación, finalmente, el nodo limpia del búfer todos los números ya procesados (tanto del espectro como de las constantes) y repite el proceso desde el principio hasta que no queden más banderas detectables. Así, se asegura de que cada paquete generado por GNU Radio se traduzca en un mensaje perfectamente formado, listo para visualizar o almacenar, sin mezclar datos de diferentes paquetes.

Las constantes mencionadas se van a organizar en una tabla según el protocolo al momento que se le ingresan datos.

Figura 8

Tabla de registro de datos ingresados por protocolo UDP o TCP.



Fecha	latitud	longitud	Azimut	Elevacion	Altitud	Descripcion	finial	fpaso	N	Datos
Esperando datos...										

Nota. En la imagen se ven los nombres de cada columna en el que se va a registrar los datos, cuando el sistema no le ingresa datos sale “Esperando datos...” hasta que empieza a recibir información, una vez recibe la información los datos se van registrando segun la columna.

5. Plotly

Cuando ya tenemos listas las ventanas espectrales completas tanto por UDP como por TCP el siguiente paso es convertir esos datos en una gráfica clara y sencilla. Para ello usamos un nodo Template en Node-RED que incluye un pequeño fragmento de código HTML y JavaScript con Plotly, una librería de gráficos interactivos. A continuación te explico, cómo funciona para cada protocolo:

5.1 Espectro UDP (Amplitud vs. Frecuencia)

Figura 9

Registro de la gráfica UDP.

```

1  <!-- TÍTULO EN AZUL -->
2  <h2 style="text-align:center; color: blue; font-weight: bold; margin-bottom: 10px;">
3  | ESPECTRO UDP
4  </h2>
5  <p id="hora-udp-paralelo" style="text-align:center;font-weight:bold;color:blue;margin:4px 0;">Última UDP: ---:---:---</p>
6
7  <!-- Muestra la hora del último espectro recibido (si existe) -->
8  <div id="last_update" style="text-align:center; font-size:16px; color:gray; margin-bottom:10px;"></div>
9
10 <!-- Contenedor del gráfico -->
11 <div id="chart_udp" style="width: 100%; height: 500px;"></div>
12
13 <script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
14 <script>
15   (function(scope){
16     let last=null;
17     const lbl=document.getElementById('hora-udp-paralelo');
18     function setTime(t){ if(lbl) lbl.textContent='Última UDP: '+t; }
19     const saved=localStorage.getItem('hora-udp-paralelo'); if(saved) setTime(saved);
20     function draw(d){
21       if(!d||!d.x||!d.y) return;
22       last=d;
23       Plotly.newPlot('chart_udp',[{x:d.x,y:d.y,type:'scatter',mode:'lines',line:{color:'blue'}}],{autosize:true,margin:{t:40,
24     }
25   scope.$watch('msg',function(msg){
26     if(!msg) return;
27     if(msg.topic==='ui_control'){setTimeout(()=>{Plotly.Plots.resize(document.getElementById('chart_udp'));},300);return;}
28     if(msg.payload.x && msg.payload.y){draw(msg.payload);}
29     else if(typeof msg.payload==='string'){localStorage.setItem('hora-udp-paralelo',msg.payload);setTime(msg.payload);}
30   });
31   })(scope);
32 </script>

```

Nota. Este es el código responsable de la gráfica de los datos ingresados por protocolo UDP.

Título y hora de actualización.

En la parte superior aparece un encabezado grande en color azul que dice “ESPECTRO UDP”.

Justo debajo, se muestra la hora en que llegó el último espectro UDP. Esa hora se guarda en el navegador para que persista aun si recargas la página.

Contenedor de la gráfica.

Hay un espacio delimitado donde Plotly dibuja la curva. Su tamaño se ajusta automáticamente al ancho de la pantalla y a la altura definida (500 px).

Dibujo del gráfico.

El código escucha de forma continua los mensajes que llegan al dashboard. Cuando detecta uno con datos válidos (dos listas: frecuencias en el eje X y valores de amplitud en el eje Y), llama a Plotly.newPlot para trazar una línea azul que une todos los puntos.

Si cambias el tamaño de la ventana, el gráfico se redibuja unos instantes después para que siga viéndose bien.

Persistencia de la hora.

Cada vez que recibe un mensaje que solo trae un texto (la hora), el script lo almacena en localStorage y actualiza la etiqueta. Así, al volver al dashboard más tarde, siempre verás la hora de la última UDP procesada.

5.2 Espectro TCP (Amplitud vs. Frecuencia)

Figura 10

Registro de la gráfica TCP.

```

1  <h2 style="text-align:center; color: red; font-weight: bold;">
2  | ESPECTRO TCP
3  </h2>
4  <p id="hora-tcp-paralelo" style="text-align:center;font-weight:bold;color:red;margin:4px 0;">Última TCP: --:--:--</p>
5
6  <div id="chart" style="width: 100%; height: 500px;"></div>
7
8  <script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
9  <script>
10 | (function(scope){
11 |   let last=null;
12 |   const lbl=document.getElementById('hora-tcp-paralelo');
13 |   function setTime(t){ if(lbl) lbl.textContent='Última TCP: '+t; }
14 |   const saved=localStorage.getItem('hora-tcp-paralelo'); if(saved) setTime(saved);
15 |   function draw(d){
16 |     if(!d||!d.x||!d.y) return;
17 |     last=d;
18 |     Plotly.newPlot('chart',[{x:d.x,y:d.y,type:'scatter',mode:'lines',line:{color:'red'}}],{autosize:true,margin:{t:40,r:4
19 |   }
20 |   scope.$watch('msg',function(msg){
21 |     if(!msg) return;
22 |     if(msg.topic==='ui_control'){setTimeout(()=>{Plotly.Plots.resize(document.getElementById('chart'));},300);return;}
23 |     if(msg.payload && msg.payload.x && msg.payload.y){draw(msg.payload);}
24 |     else if(typeof msg.payload==='string'){localStorage.setItem('hora-tcp-paralelo',msg.payload);setTime(msg.payload);}
25 |   });
26 | })(scope);
27 | </script>

```

Nota. Este es el código responsable de la gráfica de los datos ingresados por protocolo TCP.

El mecanismo es exactamente el mismo que para UDP, pero con dos diferencias de color y de etiqueta:

Gráfica roja: Cuando llegan datos por TCP, Plotly dibuja la curva con una línea roja, manteniendo la misma lógica de ejes (frecuencia en X, amplitud en Y) y el mismo comportamiento de ajuste automático.