eso-pic

# SANDRA gRPC

## STREAM PROTOCOL

**PROTOCOLO DE COMUNICACIÓN EN TIEMPO REAL**

**Versión 1.0.0** | Abril 2026


**Proyecto**

**Sandra**

Código Abierto / Sovereign Tech

CARACAS, VENEZUELA

# 1 Resumen Ejecutivo

Este documento técnico describe la implementación del protocolo gRPC Streaming en el ecosistema Sandra. El sistema utiliza comunicación bidireccional por streams para lograr procesamiento en tiempo real entre el servidor Go (Valquiria) y el motor de cálculo Rust (Loki), garantizando latencia mínima y determinismo en las operaciones de nómina.

El protocolo soporta cuatro modalidades de streaming: Server Streaming, Client Streaming, Bidirectional Streaming y Server-side Reflection, cada una optimizada para casos de uso específicos dentro del ecosistema.

**Palabras clave:** gRPC, Streaming, Protocol Buffers, Bidirectional, Sentinel, Nomina

# 2 Fundamentos de gRPC en Sandra

## 2.1 Arquitectura de Comunicación

Sandra implementa una arquitectura de comunicación híbrida que combina REST API para operaciones CRUD tradicionales con gRPC para procesamiento crítico y streaming en tiempo real.

## 2.2 Modelo de Comunicación Sandra

- **REST (Puerto 80/443)**: Autenticación, consulta de directivas, gestión de usuarios
- **gRPC (Puerto 8443)**: Procesamiento de nómina, cálculos masivos, streams financieros
- **WebSocket (WSHub)**: Notificaciones push, sincronización de estado
- **sandra-app:// (Freya)**: Protocolo propietario para desktop

## 2.3 Ventajas del Protocolo

1. **Rendimiento**: Protocol Buffers hasta 10x más rápido que JSON
2. **Tipos Fuertes**: Definición de contratos inmutables
3. **Streaming Nativo**: Soporte integrado para streams bidireccionales
4. **Generación de Código**: Clients stubs para Go, Rust, TypeScript
5. **Multiplexación**: HTTP/2 permite múltiples streams en una conexión

## 2.4 Puerto de Comunicación

```
# Configuración en sandra.ini
[server]
grpc_port=8443
```

# 3 Definición de Servicios Proto

## 3.1 Estructura del Proto

El sistema define servicios gRPC en archivos .proto que sirven como contrato único entre cliente y servidor.

```
syntax = "proto3";

package sandra.nomina;

option go_package = "github.com/proyecto-sandra/proto/nomina";

service NominaService {
    rpc CalcularNomina(Manifiesto) returns (stream ResultadoProgreso);
    rpc ProcesarFlujo(stream SolicitudNomina) returns (stream RespuestaNomina);
    rpc CargarMovimientos(stream Movimiento) returns (ResumenCarga);
    rpc ConsultarEmpleado(ConsultaEmpleadoRequest) returns (Empleado);
}

message Manifiesto {
    string ciclo = 1;
    string directiva = 2;
    string componente = 3;
    int32 partida = 4;
    bool calcular = 5;
    bool emitir = 6;
}
```

## 3.2 Manifiesto de Procesamiento

```
{
  "nombre": "Nómina Enero 2026",
  "ciclo": "2026-01",
  "directiva": "DIR-2026-01",
  "componente": "EJ",
  "partida": 201,
  "filtros": {
    "status": 201,
    "grado": ["Oficial", "Tropero", "Civil"]
  },
  "modulos": ["Base", "Directiva", "Anticipo"],
  "calcular": true,
  "emitir": true,
  "formato": "pdf"
}
```

# 4 Tipos de Stream

Sandra implementa los cuatro tipos de streaming definidos por gRPC, cada uno optimizado para escenarios específicos.

## 4.1 Server Streaming

Utilizado para operaciones donde el cliente envía una solicitud y el servidor responde con un flujo de datos progresivos. Ideal para:

- **Progreso de cálculo**: Actualizaciones en tiempo real del procesamiento
- **Reportes extensos**: Generación de nóminas con many pages
- **Logs de auditoría**: Trazabilidad continua de operaciones

```
// Cliente Go - Server Streaming
stream, err := client.CalcularNomina(ctx, &Manifiesto{
    Ciclo:      "2026-01",
    Directiva:  "DIR-2026-01",
    Componente: "EJ",
})

for {
    progreso, err := stream.Recv()
    if err == io.EOF { break }
    if err != nil { return err }
    fmt.Printf("Progreso: %d%% - %s\n",
        progreso.Porcentaje, progreso.Mensaje)
}
```

## 4.2 Client Streaming

El cliente envía múltiples mensajes sin esperar respuesta inmediata. Útil para:

- **Carga masiva**: Múltiples movimientos de nómina
- **Importación de datos**: batch processing
- **Validación distribuida**: Verificación de records

```
// Cliente Rust - Client Streaming
let mut stream = client.cargar_movimientos().await?;

for movimiento in movimientos {
    stream.send(movimiento).await?;
}

let resumen = stream.close_and_recv().await?;
println!("Cargados: {} registros", resumen.total());
println!("Errores: {}", resumen.errores());
```

### 4.3 Bidirectional Streaming

El modo más poderoso: cliente y servidor envían mensajes de forma independiente y concurrente. Usado para:

- **Procesamiento completo**: Orchestration de cálculo completo
- **Chat de control**: Comandos interactivos durante cálculo
- **Sincronización de estado**: Actualización bidireccional

```
// Bidirectional Streaming
stream, err := client.ProcesarFlujo(ctx)

// Envío concurrente de solicitudes
go func() {
    for _, req := range solicitudes {
        stream.Send(req)
        time.Sleep(100 * time.Millisecond)
    }
    stream.CloseSend()
}()

// Recepción concurrente de respuestas
go func() {
    for {
        resp, err := stream.Recv()
        if err == io.EOF { break }
        if err != nil { return err }
        procesarRespuesta(resp)
    }
}()

<-stream.Context().Done()
```

### 4.4 Server-side Reflection

Permite a clientes descubrir servicios y métodos disponibles en tiempo de ejecución.

```
# Consulta de servicios disponibles
grpcurl -plaintext localhost:8443 \
    grpc.reflection.v1.ServerReflection/ServerReflectionInfo
```

# 5 Integración con Sentinel (Loki)

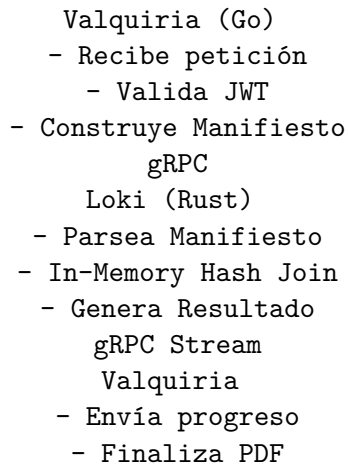
## 5.1 Comunicación gRPC

Valquiria (Go) se comunica con Loki (Rust) exclusivamente vía gRPC.

## 5.2 Beneficios de la Integración Go-Rust

- **Orquestación en Go**: Productividad y ecosistema maduro
- **Cálculo en Rust**: Rendimiento extremo y seguridad de memoria
- **Contrato Inmutable**: Proto files como fuente de verdad
- **Tipos Compartidos**: Generación automática de stubs

## 5.3 Pipeline de Procesamiento




## 5.4 Ejemplo de Llamada Completa

```
// go/sandra/core/proto/client.go
func (s *Service) ProcesarNomina(ctx context.Context, manif *NominaManifest) (*NominaResulta
    conn, err := grpc.Dial(
        "localhost:50051",
        grpc.WithTransportCredentials(insecure.NewCredentials()),
        grpc.WithBlock(),
    )
    if err != nil {
        return nil, fmt.Errorf("conexión fallida: %w", err)
    }
    defer conn.Close()

    client := nomina.NewNominaServiceClient(conn)
    stream, err := client.CalcularNomina(ctx, &Manifiesto{
        Ciclo:      manif.Ciclo,
        Directiva:  manif.Directiva,
        Componente: manif.Componente,
```

```
        Partida:    int32(manif.Partida),
        Calcular:   true,
        Emitir:     true,
    })

    var ultimoProgreso *ResultadoProgreso
    for {
        progreso, err := stream.Recv()
        if err == io.EOF { break }
        if err != nil { return nil, err }
        ultimoProgreso = progreso
        log.Printf("Progreso: %d%%", progreso.Porcentaje)
    }

    return ultimoProgreso.Resultado, nil
}
```

# 6 Manejo de Errores y Reconexión

## 6.1 Códigos de Error

Sandra utiliza códigos de error estándar de gRPC con extensiones propias:

## 6.2 Estrategia de Reintento

```
// Configuración de retry
retryPolicy := &grpcretry.RetryPolicy{
    MaxAttempts: 3,
    InitialBackoff: 100 * time.Millisecond,
    MaxBackoff: 2 * time.Second,
    BackoffMultiplier: 2.0,
    RetryableStatusCodes: []codes.Code{
        codes.Unavailable,
        codes.Internal,
    },
}
```

## 6.3 Heartbeat y Health Check

El sistema implementa verificación de salud para detectar desconexiones:

```
service Health {
    rpc Check(HealthCheckRequest) returns (HealthCheckResponse);
    rpc Watch(HealthCheckRequest) returns (stream HealthCheckResponse);
}

message HealthCheckResponse {
    enum ServingStatus {
        UNKNOWN = 0;
        SERVING = 1;
        NOT_SERVING = 2;
    }
    ServingStatus status = 1;
}
```

# 7 Security y Autenticación

## 7.1 Token JWT en Metadata

Cada llamada gRPC incluye credenciales en metadata:

```
// Cliente incluye JWT en metadata
authInterceptor := grpc.UnaryInterceptor(func(
    ctx context.Context,
    method string,
    req, reply interface{},
    cc *grpc.ClientConn,
```

```
    invoker grpc.UnaryInvoker,
) error {
    token, err := generarJWT()
    if err != nil { return err }

    ctx = metadata.NewOutgoingContext(ctx, metadata.Pairs(
        "authorization", "Bearer "+token,
    ))

    return invoker(ctx, method, req, reply, cc)
})
```

## 7.2 TLS Mutuo (mTLS)

Para producción se configura TLS bidireccional:

```
# Certificados necesarios:
# - server.crt / server.key (Valquiria)
# - client.crt / client.key (Loki)
# - ca.crt (Autoridad certificadora)

# Configuración en sandra.ini
[grpc]
tls_enabled=true
cert_file=/etc/sandra/certs/server.crt
key_file=/etc/sandra/certs/server.key
ca_file=/etc/sandra/certs/ca.crt
require_client_auth=true
```

# 8 Performance y Optimización

## 8.1 Benchmarks gRPC Sandra

- **Latencia**: <5ms para llamadas unary
- **Throughput**: >10,000 msgs/segundo en stream
- **Conexiones**: Hasta 1,000 streams concurrentes
- **Tamaño payload**: Hasta 50MB por mensaje

## 8.2 Optimizaciones Implementadas

1. **Connection Pooling**: Reutilización de conexiones
2. **Keepalive**: Mantiene conexiones activas
3. **Compression**: gzip para payloads grandes
4. **Message Size Limits**: Límites configurables
5. **Concurrent Streams**: Multiplexación HTTP/2

```
// Configuración optimizada del cliente
opts := []grpc.DialOption{
    grpc.WithTransportCredentials(creds),
    grpc.WithKeepaliveParams(keepalive.ClientParameters{
        Time:    20 * time.Second,
        Timeout: 10 * time.Second,
    }),
    grpc.WithDefaultCallOptions(
        grpc.MaxCallRecvMsgSize(50 * 1024 * 1024),
        grpc.UseCompressor(gzip.Name),
    ),
    grpc.WithConnectionPool(poolsize),
}
```

# 9 Conclusiones

El protocolo gRPC Streaming en Sandra proporciona una base sólida para la comunicación de alto rendimiento entre componentes. Las ventajas principales son:

- **Determinismo**: Contratos tipo-seguros que nunca se rompen
- **Eficiencia**: Protocol Buffers reducen bandwidth 10x vs JSON
- **Streaming Nativo**: Cuatro modos para diferentes escenarios
- **Interoperabilidad**: Clients generados para Go, Rust, TypeScript
- **Resiliencia**: Retry automático y health checks

La combinación de Go para orquestación y Rust para cálculo, comunicados por gRPC, representa la arquitectura óptima para sistemas de nómina empresariales que requieren tanto productividad como rendimiento extremo.

**Fecha:** Abril 2026
**Versión:** 1.0.0
**Proyecto Sandra** - Código Abierto

| Código | Descripción | Recuperación |
|---|---|---|
| OK | Operación exitosa | N/A |
| INVALID_ARGUMENT | Manifiesto malformado | Corregir y reintentar |
| NOT_FOUND | Empleado/directiva no existe | Verificar datos |
| INTERNAL | Error en cálculo Sentinel | Retry con backoff |
| UNAVAILABLE | Sentinel no disponible | Reconexión automática |
| CANCELLED | Cliente canceló operación | Limpiar estado |