

```

import grpc
import logging
import cv2
from typing import Iterator, List, Tuple
from src.config import AppConfig
from src.ai import SkeletonPoint
import numpy as np

# Generated gRPC modules
try:
    from src import ksl_sentence_recognition_pb2 as pb2
    from src import ksl_sentence_recognition_pb2_grpc as pb2_grpc
except ImportError:
    import ksl_sentence_recognition_pb2 as pb2
    import ksl_sentence_recognition_pb2_grpc as pb2_grpc

# Configure logging
logger = logging.getLogger(__name__)

class GrpcClient:
    def __init__(self, server_address: str):
        self.server_address = server_address
        self.channel: grpc.Channel = grpc.insecure_channel(self.server_address)
        self.stub: pb2_grpc.SequenceServiceStub = pb2_grpc.SequenceServiceStub(self.channel)
        logger.info(f"gRPC Client initialized for server: {server_address}")

    def connect(self):
        """Explicitly attempts to connect to the gRPC server (optional, channel is lazy)."""
        try:
            grpc.channel_ready_future(self.channel).result(timeout=5) # 5-second timeout
            logger.info(f"Successfully connected to gRPC server: {self.server_address}")
        except grpc.FutureTimeoutError:
            logger.error(f"Timeout connecting to gRPC server: {self.server_address}")
            raise ConnectionRefusedError(f"Could not connect to gRPC server: {self.server_address}")

    def close(self):
        """Closes the gRPC channel."""
        self.channel.close()
        logger.info("gRPC channel closed.")

    def send_stream(self, frame_iterator: Iterator[pb2.Frame]) -> Iterator[pb2.SubmitResultResponse]:
        """
        Sends a stream of Frame messages to the server.
        Input: frame_iterator (Iterator of Frame messages)
        Output: Iterator of SubmitResultResponse (server-streaming or single response from server)
        """
        try:
            # SendFrames is a client-streaming RPC, meaning the client sends a stream
            # and the server responds with a single message.
            response = self.stub.SendFrames(frame_iterator)
            yield response # Return the single response as an iterator
        except grpc.RpcError as e:
            logger.error(f"gRPC Error during SendFrames: {e.code()} - {e.details()}")
            raise

    def encode_frame(
        session_id: str,
        index: int,
        flag: int,
        image: np.ndarray,
        skeleton: List[SkeletonPoint]
    ) -> pb2.Frame:
        """
        Serializes data into a Protobuf Frame message.
        Matches C++ EncodeFrame function.
        Input:
            session_id: str
            index: int (frame index)
            flag: int
            image: np.ndarray (RGB image)
            skeleton: List[SkeletonPoint]
        Output: Frame message
        """
        if image.size == 0:
            raise ValueError("Cannot encode empty image.")

        # Convert numpy array to bytes. OpenCV stores images as (H, W, C), Protobuf needs flat bytes.
        # Ensure image is contiguous for direct byte conversion.
        if not image.flags['C_CONTIGUOUS']:
            image = np.ascontiguousarray(image)

        # Get OpenCV type equivalent for protobuf.type
        channels = 1
        if image.ndim == 3:
            channels = image.shape[2]

        depth = 0

```

```

if image.dtype == np.uint8:
    depth = cv2.CV_8U
elif image.dtype == np.int8:
    depth = cv2.CV_8S
elif image.dtype == np.uint16:
    depth = cv2.CV_16U
elif image.dtype == np.int16:
    depth = cv2.CV_16S
elif image.dtype == np.int32:
    depth = cv2.CV_32S
elif image.dtype == np.float32:
    depth = cv2.CV_32F
elif image.dtype == np.float64:
    depth = cv2.CV_64F
else:
    # Fallback or raise error? For now fallback to uint8 equivalent logic if unknown
    depth = cv2.CV_8U

# CV_MAKETYPE(depth, cn) equivalent
# ((depth) & 7) + ((cn) - 1) << 3
cv_type_equivalent = (depth & 7) + ((channels - 1) << 3)

frame_data = image.tobytes()

proto_pose_points = []
for sp in skeleton:
    proto_pose_points.append(pb2.Point3(x=sp.x, y=sp.y, z=sp.z))

frame = pb2.Frame(
    session_id=session_id,
    index=index,
    flag=flag,
    width=image.shape[1], # Width is columns
    height=image.shape[0], # Height is rows
    type=cv_type_equivalent, # Match C++ type logic
    data=frame_data,
    pose_points=proto_pose_points
)
return frame

```