Sistema de Detección de Ciberataques DDoS por Análisis de Tráfico de Red

Master en IA sobre el sector de la Energía y las infraestructuras

Autor: Ramiro Bueno Martínez

Fecha: 06/08/2025

Version: R200

Descripcion: PARTE IV: Implementación Distribuida basada en MAS

Analisis de Tráfico de Red en Ataques DDoS

El ataque de denegación de servicio (DDoS) es una amenaza a la seguridad de la red que tiene como objetivo agotar las redes de destino con tráfico malicioso. Aunque se han diseñado muchos métodos estadísticos para la detección de ataques DDoS, diseñar un detector en tiempo real con techos computacionales bajos sigue siendo una de las principales preocupaciones. Por otro lado, la evaluación de nuevos algoritmos y técnicas de detección se basa en gran medida en la existencia de conjuntos de datos bien diseñados.

Referencias estandares normativos

ISO/IEC FDIS 23053 Framework for Artificial Intelligence (AI) Systems Using Machine Learning (ML) Note: Under development ISO/IEC FDIS 23053

ISO/IEC CD 8183 Information technology — Artificial intelligence — Data life cycle framework Note: Under development ISO/IEC CD 8183

ISO/IEC DIS 24668 Information technology — Artificial intelligence — Process management framework for big data analytics Note: Under development ISO/IEC DIS 24668

ISO/IEC CD 5338 Information technology — Artificial intelligence — AI system life cycle processes Note: Under developmentISO/IEC CD 5338

Objetivo Principal:

El objetivo principal implementar un sistema distribuido de detección de posibles ataques DDoS a través de la Red mediante un sistema basado en agentes autonomos y multiagente

Agente BDI

Un Agente BDI (Beliefs, Desires, Intentions) es un modelo de agente racional que imita la toma de decisiones humana. Representa el estado mental de un agente de software y le permite razonar de manera autónoma en entornos dinámicos. - Creencias (Beliefs): El conocimiento o la información que el agente tiene sobre su entorno. Son hechos o percepciones que el agente considera verdaderos. - Deseos (Desires): Son los objetivos, tareas o estados que el agente quiere alcanzar. Representan el propósito o la motivación del agente. - Intenciones (Intentions): Son los planes de acción que el agente ha seleccionado para perseguir sus deseos. Son los compromisos del agente para ejecutar una secuencia de pasos.

El modelo BDI proporciona un marco estructurado para que el agente reciba información (creencias), evalúe sus metas (deseos) y decida qué hacer (intenciones).

Agente BDI y Conciencia Situacional para la Detección de DDoS

La Conciencia Situacional (SA) es la capacidad de percibir, comprender y proyectar el estado de un entorno. Para un Agente BDI, la Conciencia Situacional es el resultado de un ciclo continuo donde las creencias se actualizan a partir de datos del entorno, lo que a su vez impulsa los deseos y las intenciones del agente.

En el contexto de la detección de ataques DDoS, un agente BDI dota de inteligencia al sistema de defensa de la siguiente manera: - Percepción y Creencias: El agente adquiere conciencia situacional del tráfico de red. En lugar de procesar los datos de red por sí mismo, utiliza un modelo de Machine Learning o Deep Learning preentrenado. Este modelo, entrenado con un conjunto de datos como el CICDDoS2019, actúa como el "sentido" del agente. Sus predicciones sobre la naturaleza del tráfico (normal, ataque) se convierten en las creencias del agente. Por ejemplo, si el modelo detecta un patrón de tráfico malicioso, el agente establece la creencia: "Se está produciendo un ataque DoS de tipo UDP Flood." - Comprensión y Deseos: A partir de sus creencias, el agente comprende la situación y activa sus deseos. El deseo principal es mantener la disponibilidad y la estabilidad de la red. Esto se traduce en objetivos como "mitigar el ataque", "minimizar la pérdida de paquetes" y "proteger los recursos críticos". - Proyección e Intenciones:

Con sus deseos claros y sus creencias actualizadas, el agente formula un plan de acción (intención). Este plan puede ser simple o complejo y se ejecuta para alcanzar sus deseos. Por ejemplo, el agente puede decidir:

```
- Bloquear las direcciones IP de origen sospechosas.
- Limitar la tasa de tráfico (rate-limiting) en un puerto específico.
- Generar una alerta y notificar a los administradores del sistema.
```

```
import os
import pandas as pd
import numpy as np
from datetime import datetime
import time
import threading
import queue
import joblib
import pickle
import logging
from typing import Dict, List, Any, Optional
from enum import Enum
from dataclasses import dataclass, field
import subprocess
from io import StringIO
import csv
from sklearn.utils.validation import check_array
from pathlib import Path  # Mejor manejo de rutas

import warnings
warnings.filterwarnings("ignore", category=UserWarning, message="X does not have valid feature names")

# Configuración de logging
logging.basicConfig(
```

```
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

class ThreatLevel(Enum):
    """Niveles de amenaza detectados"""
    NONE = "none"
    LOW = "low"
    MEDIUM = "medium"
    HIGH = "high"
    CRITICAL = "critical"

class AgentState(Enum):
    """Estados del agente BDI"""
    IDLE = "idle"
    CAPTURING = "capturing"
    PROCESSING = "processing"
    ANALYZING = "analyzing"
    RESPONDING = "responding"

@dataclass
class Belief:
    """Representa una creencia del agente"""
    key: str
    value: Any
    confidence: float
    timestamp: datetime = field(default_factory=datetime.now)

@dataclass
class Desire:
```

```
    """Representa un deseo/objetivo del agente"""
    goal: str
    priority: int
    active: bool = True

@dataclass
class Intention:
    """Representa una intención/plan del agente"""
    action: str
    parameters: Dict[str, Any]
    expected_outcome: str
    status: str = "pending"

@dataclass
class ThreatAlert:
    """Representa una alerta de amenaza"""
    timestamp: datetime
    threat_level: ThreatLevel
    attack_type: Optional[str]
    confidence: float
    source_ip: str
    destination_ip: str
    details: Dict[str, Any]

class NetworkCapture:
    """Módulo de captura de red basado en el código original"""

    def __init__(self, interface: str = "wlp1s0"):
        self.interface = interface
        self.output_dir_pcap = "./pcap"
        self.output_dir_csv = "./csv"
```

```
        self.output_dir_prod = "./prod"
        self._setup_directories()

    def _setup_directories(self):
        """Crear directorios necesarios"""
        for directory in [self.output_dir_pcap, self.output_dir_csv, self.output_dir_prod]:
            if not os.path.exists(directory):
                os.makedirs(directory)

    def capture_traffic(self, duration: int) -> str:
        """Captura tráfico de red y retorna la ruta del archivo pcap"""
        now = datetime.now()
        timestamp = now.strftime("%d%m%Y-%H%M%S")
        #pcap_output_path = os.path.join(self.output_dir_pcap, f"{timestamp}.pcap")
        # Usamos Path para garantizar compatibilidad multiplataforma
        pcap_output_path = Path(self.output_dir_pcap) / f"{timestamp}.pcap"
        # Convertimos a string y normalizamos los separadores
        pcap_output_path = os.path.normpath(pcap_output_path)

        command = f"tshark -i {self.interface} -a duration:{duration} -w {pcap_output_path} >> standard.out"
        logger.info(f"Ejecutando captura: {command}")

        result = os.system(command)
        if result != 0:
            raise Exception(f"Error en la captura de tráfico: código {result}")

        return pcap_output_path

    import os

    def extract_flow_stats_with_tshark(self, pcap_path: str, output_csv: str) -> Optional[str]:
```

```
    """Extrae estadísticas de flujo similares a CICFlowMeter usando tshark + pandas.

    Args:
        pcap_path: Ruta al archivo .pcap.
        output_csv: Ruta de salida para el CSV.

    Returns:
        Ruta del CSV generado o None si hay error.
    """
    # Lista de columnas esperadas (25 features)
    expected_columns = [
        'Source Port', 'Destination Port', 'Protocol', 'Flow Duration',
        'Total Fwd Packets', 'Total Backward Packets', 'Total Length of Fwd Packets',
        'Total Length of Bwd Packets', 'Fwd Packet Length Max', 'Fwd Packet Length Min',
        'Fwd Packet Length Mean', 'Fwd Packet Length Std', 'Bwd Packet Length Max',
        'Bwd Packet Length Min', 'Bwd Packet Length Mean', 'Bwd Packet Length Std',
        'Flow IAT Min', 'Flow IAT Max', 'Flow IAT Mean', 'Flow IAT Std',
        'Fwd IAT Total', 'Fwd IAT Mean', 'Fwd IAT Std', 'Flow Packets/s',
        'Flow Bytes/s'
    ]

    # 1. Extraer datos crudos con tshark
    command = [
        "tshark",
        "-r", pcap_path,
        "-T", "fields",
        "-e", "frame.time_epoch",
        "-e", "ip.src",
        "-e", "ip.dst",
        "-e", "tcp.srcport", "-e", "tcp.dstport",
        #"-e", "udp.srcport", "-e", "udp.dport",
```

```
        "-e", "udp.srcport", "-e", "udp.dstport",  # <-- Línea corregida
        "-e", "ip.proto",
        "-e", "frame.len",
        "-E", "header=y",
        "-E", "separator=,",
        "-E", "quote=d",
    ]

    try:
        # Ejecutar tshark y cargar datos
        result = subprocess.run(command, capture_output=True, text=True, check=True)
        df = pd.read_csv(StringIO(result.stdout))

        # 2. Preprocesamiento de datos
        df.columns = ["timestamp", "src_ip", "dst_ip",
                      "tcp_sport", "tcp_dport", "udp_sport", "udp_dport",
                      "proto", "pkt_len"]

        # Limpieza y conversión de tipos
        df = df.fillna(0)
        df["src_port"] = df["tcp_sport"].fillna(df["udp_sport"]).astype(int)
        df["dst_port"] = df["tcp_dport"].fillna(df["udp_dport"]).astype(int)
        df["proto"] = df["proto"].astype(int)

        # 3. Agrupar por flujo
        df_flows = df.groupby(["src_ip", "dst_ip", "src_port", "dst_port", "proto"])

        # 4. Calcular estadísticas
        flow_stats = []
        for name, group in df_flows:
            fwd = group[group["src_ip"] == name[0]]
```

```
            bwd = group[group["src_ip"] == name[1]]

            flow_duration = max(group["timestamp"].max() - group["timestamp"].min(), 1e-6)

            # --- CORRECCIÓN: Definir 'flow_packets_rate' y 'flow_bytes_rate' aquí ---
            total_packets = len(group)
            total_bytes = group["pkt_len"].sum()

            flow_packets_rate = total_packets / flow_duration
            flow_bytes_rate = total_bytes / flow_duration
            # Calcular las diferencias de tiempo entre paquetes consecutivos en todo el flujo
            # La primera diferencia es NaN, por lo que la eliminamos (.iloc[1:])
            diffs = group["timestamp"].sort_values().diff().iloc[1:]

            # Calcular las diferencias de tiempo entre paquetes consecutivos en el flujo de avance (fwd)
            fwd_diffs = fwd["timestamp"].sort_values().diff().iloc[1:]
            stats = {
                "Source Port": int(name[2]),
                "Destination Port": int(name[3]),
                "Protocol": int(name[4]),
                "Flow Duration": float(flow_duration),
                "Total Fwd Packets": int(len(fwd)),
                "Total Backward Packets": int(len(bwd)),
                "Total Length of Fwd Packets": int(fwd["pkt_len"].sum()),
                "Total Length of Bwd Packets": int(bwd["pkt_len"].sum() if len(bwd) > 0 else 0),
                "Fwd Packet Length Max": int(fwd["pkt_len"].max()) if len(fwd) > 0 else 0,
                "Fwd Packet Length Min": int(fwd["pkt_len"].min()) if len(fwd) > 0 else 0,
                "Fwd Packet Length Mean": float(fwd["pkt_len"].mean()) if len(fwd) > 0 else 0.0,
                "Fwd Packet Length Std": float(fwd["pkt_len"].std()) if len(fwd) > 1 else 0.0,
                "Bwd Packet Length Max": int(bwd["pkt_len"].max()) if len(bwd) > 0 else 0,
                "Bwd Packet Length Min": int(bwd["pkt_len"].min()) if len(bwd) > 0 else 0,
```

```
                "Bwd Packet Length Mean": float(bwd["pkt_len"].mean()) if len(bwd) > 0 else 0.0,
                "Bwd Packet Length Std": float(bwd["pkt_len"].std()) if len(bwd) > 1 else 0.0,
                "Flow IAT Min": float(diffs.min()) if len(diffs) > 0 else 0.0,
                "Flow IAT Max": float(diffs.max()) if len(diffs) > 0 else 0.0,
                "Flow IAT Mean": float(diffs.mean()) if len(diffs) > 0 else 0.0,
                "Flow IAT Std": float(diffs.std()) if len(diffs) > 1 else 0.0,
                "Fwd IAT Total": float(fwd_diffs.sum()) if len(fwd_diffs) > 0 else 0.0,
                "Fwd IAT Mean": float(fwd_diffs.mean()) if len(fwd_diffs) > 0 else 0.0,
                "Fwd IAT Std": float(fwd_diffs.std()) if len(fwd_diffs) > 1 else 0.0,
                "Flow Packets/s": float(flow_packets_rate),
                "Flow Bytes/s": float(flow_bytes_rate)
            }
            flow_stats.append(stats)
        # 5. Crear DataFrame y asegurar las 25 columnas
        df_result = pd.DataFrame(flow_stats).fillna(0)

        # Verificar columnas antes de guardar
        missing_cols = set(expected_columns) - set(df_result.columns)
        if missing_cols:
            raise ValueError(f"Columnas faltantes: {missing_cols}")

        # Guardar CSV con formato controlado
        df_result[expected_columns].to_csv(
            output_csv,
            index=False,
            float_format='%.6f',
            quoting=csv.QUOTE_NONNUMERIC
        )
        return output_csv

    except subprocess.CalledProcessError as e:
```

```
            print(f"Error en tshark (¿está instalado?): {e.stderr}")
        except Exception as e:
            print(f"Error inesperado: {str(e)}")
        return None

    def convert_pcap_to_csv(self, pcap_path: str, output_dir: str) -> Optional[str]:
        """Convierte un archivo PCAP a CSV usando tshark (Wireshark).

        Args:
            pcap_path: Ruta completa al archivo .pcap.
            output_dir: Directorio donde se guardará el CSV.

        Returns:
            Ruta del archivo CSV generado o None si hay error.

        Raises:
            Exception: Si tshark no está instalado o falla la conversión.
        """
        # Validar entrada
        if not os.path.isfile(pcap_path):
            raise FileNotFoundError(f"Archivo PCAP no encontrado: {pcap_path}")

        os.makedirs(output_dir, exist_ok=True)

        # Nombre del CSV (mismo nombre que el PCAP pero con extensión .csv)
        timestamp = os.path.splitext(os.path.basename(pcap_path))[0]
        csv_output_path = os.path.join(output_dir, f"{timestamp}_tshark.csv")

        # Campos a extraer (personaliza según necesidades)
        fields = [
            "frame.time", "ip.src", "ip.dst", "ip.proto",
```

```
        "tcp.srcport", "tcp.dstport", "udp.srcport", "udp.dstport",
        "frame.len", "tcp.flags", "udp.length", "ip.ttl"
    ]

    # Comando tshark
    command = [
        "tshark",
        "-r", pcap_path,  # Archivo de entrada
        "-T", "fields",
        *[f"-e {field}" for field in fields],
        "-E", "header=y",  # Incluir cabeceras
        "-E", "separator=,",  # CSV
        "-E", "quote=d",  # Comillas dobles
    ]

    # Ejecutar y redirigir salida al archivo CSV
    try:
        with open(csv_output_path, "w") as csv_file:
            subprocess.run(
                command,
                stdout=csv_file,
                stderr=subprocess.PIPE,
                check=True,
                text=True
            )
        return csv_output_path

    except subprocess.CalledProcessError as e:
        error_msg = f"Error en tshark (código {e.returncode}): {e.stderr}"
        raise Exception(error_msg) from e
    except FileNotFoundError:
```

```
            raise Exception("tshark no está instalado. Instala Wireshark primero.")

    def preprocess_for_classification(self, df: pd.DataFrame) -> pd.DataFrame:
        """Elimina columnas no necesarias y asegura 25 features"""
        # Eliminar columna 'Unnamed: 0' si existe
        if 'Unnamed: 0' in df.columns:
            df = df.drop(columns=['Unnamed: 0'])

        # Verificar que tengamos exactamente 25 columnas
        expected_columns = [
            'Source Port', 'Destination Port', 'Protocol', 'Flow Duration',
            'Total Fwd Packets', 'Total Backward Packets', 'Total Length of Fwd Packets',
            'Total Length of Bwd Packets', 'Fwd Packet Length Max', 'Fwd Packet Length Min',
            'Fwd Packet Length Mean', 'Fwd Packet Length Std', 'Bwd Packet Length Max',
            'Bwd Packet Length Min', 'Bwd Packet Length Mean', 'Bwd Packet Length Std',
            'Flow IAT Min', 'Flow IAT Max', 'Flow IAT Mean', 'Flow IAT Std',
            'Fwd IAT Total', 'Fwd IAT Mean', 'Fwd IAT Std', 'Flow Packets/s',
            'Flow Bytes/s'
        ]

        # Reordenar columnas si es necesario
        return df[expected_columns]

    def validate_columns(self, df: pd.DataFrame) -> pd.DataFrame:
        expected_columns = [
            'Source Port', 'Destination Port', 'Protocol', 'Flow Duration',
            'Total Fwd Packets', 'Total Backward Packets', 'Total Length of Fwd Packets',
            'Total Length of Bwd Packets', 'Fwd Packet Length Max', 'Fwd Packet Length Min',
            'Fwd Packet Length Mean', 'Fwd Packet Length Std', 'Bwd Packet Length Max',
            'Bwd Packet Length Min', 'Bwd Packet Length Mean', 'Bwd Packet Length Std',
            'Flow IAT Min', 'Flow IAT Max', 'Flow IAT Mean', 'Flow IAT Std',
```

```
            'Fwd IAT Total', 'Fwd IAT Mean', 'Fwd IAT Std', 'Flow Packets/s',
            'Flow Bytes/s'
        ]

        # Verificar si todas las columnas esperadas existen
        missing = set(expected_columns) - set(df.columns)
        if missing:
            raise ValueError(f"Columnas faltantes en el DataFrame: {missing}")

        return df[expected_columns]

    def reduce_dataset(self, csv_path: str) -> str:
        """Reduce el dataset a las columnas necesarias"""
        try:
            pcap_data = pd.read_csv(csv_path)
            pcap_data = self.validate_columns(pcap_data)
            logger.info(f"Dataset cargado: {csv_path}, shape: {pcap_data.shape}")

            # Campos seleccionados del código original
            selected_fields = [
                'Source Port', 'Destination Port', 'Protocol', 'Flow Duration',
                'Total Fwd Packets', 'Total Backward Packets', 'Total Length of Fwd Packets',
                'Total Length of Bwd Packets', 'Fwd Packet Length Max', 'Fwd Packet Length Min',
                'Fwd Packet Length Mean', 'Fwd Packet Length Std', 'Bwd Packet Length Max',
                'Bwd Packet Length Min', 'Bwd Packet Length Mean', 'Bwd Packet Length Std',
                'Flow IAT Min', 'Flow IAT Max', 'Flow IAT Mean', 'Flow IAT Std',
                'Fwd IAT Total', 'Fwd IAT Mean', 'Fwd IAT Std', 'Flow Packets/s',
                'Flow Bytes/s'
            ]

            # Verificar que las columnas existen
```

```
            available_fields = [field for field in selected_fields if field in pcap_data.columns]
            if not available_fields:
                logger.warning("No se encontraron las columnas esperadas, usando todas las disponibles")
                selected_data = pcap_data
            else:
                logger.info("Los atributos de las trazas de red son correctos")
                selected_data = pcap_data[available_fields]

            # Mapeo de nombres de columnas
            column_name_mapping = {
                'src_ip': 'Source IP',
                'dst_ip': 'Destination IP',
                'src_port': 'Source Port',
                'dst_port': 'Destination Port',
                'protocol': 'Protocol',
                'timestamp': 'Timestamp',
                'flow_duration': 'Flow Duration',
                'tot_fwd_pkts': 'Total Fwd Packets',
                'tot_bwd_pkts': 'Total Backward Packets'
            }

            # Renombrar columnas disponibles
            rename_mapping = {k: v for k, v in column_name_mapping.items() if k in selected_data.columns}
            selected_data = selected_data.rename(columns=rename_mapping)

            # Guardar dataset reducido
            timestamp = os.path.splitext(os.path.basename(csv_path))[0]
            #reduced_path = os.path.join(self.output_dir_prod, f"{timestamp}_reduced.csv")

            # Usamos Path para garantizar compatibilidad multiplataforma
            reduced_path = Path(self.output_dir_prod) / f"{timestamp}_reduced.csv"
```

```
            # Convertimos a string y normalizamos los separadores
            reduced_path = os.path.normpath(reduced_path)

            selected_data.to_csv(reduced_path, index=True)

            logger.info(f"Dataset reducido guardado: {reduced_path}")
            return reduced_path

        except Exception as ex:
            logger.error(f"Error reduciendo dataset: {ex}")
            raise

class MLClassifier:
    """Módulo de clasificación usando modelos ML"""

    def __init__(self, model_path: str):
        self.model_path = model_path
        self.model = None
        self.feature_columns = None
        self.load_model()

    def load_model(self):
        """Carga el modelo ML desde archivo"""
        try:
            if self.model_path.endswith('.joblib'):
                self.model = joblib.load(self.model_path)
            elif self.model_path.endswith('.pkl') or self.model_path.endswith('.pickle'):
                with open(self.model_path, 'rb') as f:
                    self.model = pickle.load(f)
            else:
                raise ValueError("Formato de modelo no soportado. Use .joblib, .pkl o .pickle")
```

```
            logger.info(f"Modelo cargado exitosamente: {self.model_path}")

        except Exception as ex:
            logger.error(f"Error cargando modelo: {ex}")
            raise

    def prepare_features(self, data: pd.DataFrame) -> np.ndarray:
        """Prepara características asegurando compatibilidad con los feature names del modelo"""
        try:
            # 1. Limpieza inicial silenciosa
            data = data.drop(columns=['Unnamed: 0'], errors='ignore')

            # 2. Identificar/validar columnas necesarias
            required_features = [
                'Source Port', 'Destination Port', 'Protocol', 'Flow Duration',
                'Total Fwd Packets', 'Total Backward Packets', 'Total Length of Fwd Packets',
                'Total Length of Bwd Packets', 'Fwd Packet Length Max', 'Fwd Packet Length Min',
                'Fwd Packet Length Mean', 'Fwd Packet Length Std', 'Bwd Packet Length Max',
                'Bwd Packet Length Min', 'Bwd Packet Length Mean', 'Bwd Packet Length Std',
                'Flow IAT Min', 'Flow IAT Max', 'Flow IAT Mean', 'Flow IAT Std',
                'Fwd IAT Total', 'Fwd IAT Mean', 'Fwd IAT Std', 'Flow Packets/s',
                'Flow Bytes/s'
            ]

            # 3. Verificación estricta de columnas
            missing = set(required_features) - set(data.columns)
            if missing:
                raise ValueError(f"Features faltantes: {missing}")

            # 4. Orden exacto y conversión
```

```
            features = data[required_features].fillna(0)

            # 5. Conversión que preserva feature names
            from sklearn.utils.validation import check_array
            return check_array(
                features,
                accept_sparse=False,  # Más eficiente para datos densos
                ensure_all_finite=True,
                dtype=np.float64
            )

        except Exception as ex:
            logger.error(f"Error preparando features: {str(ex)}")
            raise

    def classify(self, data: pd.DataFrame) -> List[Dict[str, Any]]:
        """Clasifica los datos y retorna predicciones"""
        try:
            if self.model is None:
                raise ValueError("Modelo no cargado")

            features = self.prepare_features(data)
            ## Correcciones de los Warnings

            predictions = self.model.predict(features)

            # Si el modelo soporta probabilidades
            try:
                probabilities = self.model.predict_proba(features)
                max_probs = np.max(probabilities, axis=1)
            except:
```

```
                max_probs = np.ones(len(predictions))  # Confianza por defecto

            results = []
            for i, (pred, prob) in enumerate(zip(predictions, max_probs)):
                results.append({
                    'row_index': i,
                    'prediction': pred,
                    'confidence': float(prob),
                    'is_attack': pred != 0 if isinstance(pred, (int, np.integer)) else 'attack' in str(pred).lower()
                })

            return results

        except Exception as ex:
            logger.error(f"Error en clasificación: {ex}")
            raise

class BDINetworkAgent:
    """Agente BDI principal para detección de ataques de red"""

    def __init__(self, interface: str = "wlp1s0", model_path: str = "model.joblib"):
        self.interface = interface
        self.model_path = model_path

        # Componentes del agente
        self.network_capture = NetworkCapture(interface)
        self.ml_classifier = MLClassifier(model_path)

        # Estado BDI
        self.beliefs: Dict[str, Belief] = {}
        self.desires: List[Desire] = []
```

```
        self.intentions: queue.Queue = queue.Queue()
        self.current_state = AgentState.IDLE

        # Configuración
        self.capture_duration = 30
        self.monitoring_interval = 60
        self.threat_threshold = 0.7

        # Cola de alertas
        self.alerts: queue.Queue = queue.Queue()

        # Control de threads
        self.running = False
        self.monitoring_thread = None

        self._initialize_desires()

    def _initialize_desires(self):
        """Inicializa los deseos/objetivos del agente"""
        self.desires = [
            Desire("monitor_network", priority=1),
            Desire("detect_threats", priority=2),
            Desire("respond_to_attacks", priority=3),
            Desire("maintain_security", priority=4)
        ]

    def update_belief(self, key: str, value: Any, confidence: float = 1.0):
        """Actualiza una creencia del agente"""
        self.beliefs[key] = Belief(key, value, confidence)
        logger.debug(f"Creencia actualizada: {key} = {value} (confianza: {confidence})")
```

```python
    def add_intention(self, action: str, parameters: Dict[str, Any], expected_outcome: str):
        """Añade una nueva intención a la cola"""
        intention = Intention(action, parameters, expected_outcome)
        self.intentions.put(intention)
        logger.debug(f"Intención añadida: {action}")

    def process_network_data(self, csv_path: str) -> List[ThreatAlert]:
        """Procesa datos de red y genera alertas"""
        try:
            self.current_state = AgentState.ANALYZING

            # Cargar datos
            data = pd.read_csv(csv_path)
            _description = data.describe()
            logger.info(f"(process_network_data) Procesando {len(data)} registros de red")


            # Clasificar con ML
            predictions = self.ml_classifier.classify(data)

            # Generar alertas
            alerts = []
            attack_count = 0

            for pred in predictions:
                if pred['is_attack'] and pred['confidence'] >= self.threat_threshold:
                    attack_count += 1

                    # Extraer información del registro
                    row = data.iloc[pred['row_index']]
```

```
                # Determinar nivel de amenaza
                if pred['confidence'] >= 0.9:
                    threat_level = ThreatLevel.CRITICAL
                elif pred['confidence'] >= 0.8:
                    threat_level = ThreatLevel.HIGH
                elif pred['confidence'] >= 0.7:
                    threat_level = ThreatLevel.MEDIUM
                else:
                    threat_level = ThreatLevel.LOW

                alert = ThreatAlert(
                    timestamp=datetime.now(),
                    threat_level=threat_level,
                    attack_type=str(pred['prediction']),
                    confidence=pred['confidence'],
                    source_ip=row.get('Source IP', 'unknown'),
                    destination_ip=row.get('Destination IP', 'unknown'),
                    details={
                        'row_index': pred['row_index'],
                        'prediction': pred['prediction'],
                        'raw_data': row.to_dict()
                    }
                )

                alerts.append(alert)
                self.alerts.put(alert)

        # Actualizar creencias
        self.update_belief("last_analysis_time", datetime.now())
        self.update_belief("total_flows_analyzed", len(data))
        self.update_belief("attacks_detected", attack_count)
```

```
            self.update_belief("attack_rate", attack_count / len(data) if len(data) > 0 else 0)

            logger.info(f"Análisis completado: {attack_count} ataques detectados de {len(data)} flujos")

            return alerts

        except Exception as ex:
            logger.error(f"Error procesando datos de red: {ex}")
            return []

    # Antes de procesar, verifica y ajusta las columnas

    def capture_and_analyze(self):
        """Captura tráfico y lo analiza"""
        try:

            self.current_state = AgentState.CAPTURING
            logger.info("Iniciando captura de tráfico...")

            # Capturar tráfico
            _pcap_path = self.network_capture.capture_traffic(self.capture_duration)
            _output_path = './output.csv'
            self.current_state = AgentState.PROCESSING
            logger.info("Procesando captura...")

            csv_path = Path(self.network_capture.extract_flow_stats_with_tshark(pcap_path=_pcap_path, output_csv=_outp

            reduced_csv_path = self.network_capture.reduce_dataset(csv_path)

            # Analizar con ML
            alerts = self.process_network_data(reduced_csv_path)
```

```
            # Responder a amenazas críticas
            critical_alerts = [a for a in alerts if a.threat_level == ThreatLevel.CRITICAL]
            if critical_alerts:
                self.add_intention("respond_to_critical_threat",
                                   {"alerts": critical_alerts},
                                   "threat_mitigated")

            self.current_state = AgentState.IDLE

        except Exception as ex:
            logger.error(f"Error en captura y análisis: {ex}")
            self.current_state = AgentState.IDLE

    def execute_intentions(self):
        """Ejecuta las intenciones pendientes"""
        while self.running:
            try:
                intention = self.intentions.get(timeout=1)

                if intention.action == "respond_to_critical_threat":
                    self._respond_to_threats(intention.parameters["alerts"])
                    intention.status = "completed"

                self.intentions.task_done()

            except queue.Empty:
                continue
            except Exception as ex:
                logger.error(f"Error ejecutando intención: {ex}")
```

```python
    def _respond_to_threats(self, alerts: List[ThreatAlert]):
        """Responde a amenazas críticas"""
        self.current_state = AgentState.RESPONDING

        for alert in alerts:
            logger.warning(f"AMENAZA CRÍTICA DETECTADA: {alert.attack_type} desde {alert.source_ip}")
            logger.warning(f"Confianza: {alert.confidence:.2%}")
            logger.warning(f"Detalles: {alert.details}")

            # Aquí podrías implementar respuestas automáticas como:
            # - Bloquear IP en firewall
            # - Enviar notificaciones
            # - Registrar en SIEM
            # - Etc.

        self.current_state = AgentState.IDLE

    def monitoring_loop(self):
        """Bucle principal de monitoreo"""
        while self.running:
            try:
                logger.info("Iniciando ciclo de monitoreo...")
                self.capture_and_analyze()

                # Esperar antes del siguiente ciclo
                time.sleep(self.monitoring_interval)

            except Exception as ex:
                logger.error(f"Error en bucle de monitoreo: {ex}")
                time.sleep(10)  # Pausa antes de reintentar
```

```python
    def start_monitoring(self):
        """Inicia el monitoreo continuo"""
        if self.running:
            logger.warning("El monitoreo ya está en ejecución")
            return

        self.running = True
        logger.info("Iniciando agente BDI de seguridad de red...")

        # Iniciar thread de monitoreo
        self.monitoring_thread = threading.Thread(target=self.monitoring_loop)
        self.monitoring_thread.daemon = True
        self.monitoring_thread.start()

        # Iniciar thread de ejecución de intenciones
        intentions_thread = threading.Thread(target=self.execute_intentions)
        intentions_thread.daemon = True
        intentions_thread.start()

        logger.info("Agente BDI iniciado correctamente")

    def stop_monitoring(self):
        """Detiene el monitoreo"""
        if not self.running:
            logger.warning("El monitoreo no está en ejecución")
            return

        logger.info("Deteniendo agente BDI...")
        self.running = False

        if self.monitoring_thread:
```

```
            self.monitoring_thread.join(timeout=5)

        logger.info("Agente BDI detenido")

    def get_status(self) -> Dict[str, Any]:
        """Obtiene el estado actual del agente"""
        return {
            "state": self.current_state.value,
            "beliefs": {k: {"value": v.value, "confidence": v.confidence, "timestamp": v.timestamp}
                        for k, v in self.beliefs.items()},
            "active_desires": [d.goal for d in self.desires if d.active],
            "pending_intentions": self.intentions.qsize(),
            "pending_alerts": self.alerts.qsize()
        }

    def get_recent_alerts(self, limit: int = 10) -> List[ThreatAlert]:
        """Obtiene las alertas recientes"""
        alerts = []
        temp_alerts = []

        # Extraer alertas de la cola
        while not self.alerts.empty() and len(alerts) < limit:
            alert = self.alerts.get()
            alerts.append(alert)
            temp_alerts.append(alert)

        # Devolver alertas a la cola
        for alert in temp_alerts:
            self.alerts.put(alert)

        return alerts
```

```python
def main():
    """Función principal para demostrar el uso del agente"""

    # Configuración
    INTERFACE = "eth0"  # Interfaz de red
    MODEL_PATH = "./models/ada_boost_training_exp_attack_tcp.joblib"  # Ruta a tu modelo ML

    # Verificar que existe el modelo
    if not os.path.exists(MODEL_PATH):
        logger.error(f"Modelo no encontrado: {MODEL_PATH}")
        logger.info("Por favor, proporciona un modelo válido (.joblib, .pkl, .pickle)")
        return

    try:
        # Crear y configurar el agente
        agent = BDINetworkAgent(interface=INTERFACE, model_path=MODEL_PATH)

        # Configurar parámetros
        agent.capture_duration = 30  # 30 segundos de captura
        agent.monitoring_interval = 120  # Analizar cada 2 minutos
        agent.threat_threshold = 0.6  # Umbral de confianza para alertas

        # Iniciar monitoreo
        agent.start_monitoring()

        # Monitoreo interactivo
        print("\n=== AGENTE BDI DE SEGURIDAD DE RED ===")
        print("Comandos disponibles:")
        print("  status  - Ver estado del agente")
        print("  alerts  - Ver alertas recientes")
```

```
        print("  capture - Forzar captura manual")
        print("  quit    - Salir")
        print("=====================================\n")

        while True:
            try:
                command = input(">>> ").strip().lower()

                if command == "quit":
                    break
                elif command == "status":
                    status = agent.get_status()
                    print(f"\nEstado: {status['state']}")
                    print(f"Creencias activas: {len(status['beliefs'])}")
                    print(f"Intenciones pendientes: {status['pending_intentions']}")
                    print(f"Alertas pendientes: {status['pending_alerts']}")
                    for belief, data in status['beliefs'].items():
                        print(f"  {belief}: {data['value']} (confianza: {data['confidence']:.2f})")

                elif command == "alerts":
                    alerts = agent.get_recent_alerts(5)
                    if alerts:
                        print(f"\n=== ALERTAS RECIENTES ({len(alerts)}) ===")
                        for alert in alerts:
                            print(f"[{alert.timestamp.strftime('%H:%M:%S')}] "
                                  f"{alert.threat_level.value.upper()}: {alert.attack_type}")
                            print(f"  Desde: {alert.source_ip} -> {alert.destination_ip}")
                            print(f"  Confianza: {alert.confidence:.2%}")
                    else:
                        print("No hay alertas recientes")
```

```
                elif command == "capture":
                    print("Forzando captura manual...")
                    threading.Thread(target=agent.capture_and_analyze).start()

                else:
                    print("Comando no reconocido")

            except KeyboardInterrupt:
                break
            except Exception as ex:
                logger.error(f"Error en comando: {ex}")

        # Detener el agente
        agent.stop_monitoring()
        print("Agente detenido.")

    except Exception as ex:
        logger.error(f"Error iniciando agente: {ex}")

if __name__ == "__main__":
    main()
```

Integración de Agentes BDI en Contenedores Docker

La contenerización con Docker ofrece una solución robusta y reproducible para desplegar agentes BDI (Beliefs, Desires, Intentions). Al encapsular el agente y todas sus dependencias en un entorno aislado, garantizamos que el agente funcione de manera consistente en cualquier sistema.

1. Requisitos y Dependencias Para construir el contenedor, primero necesitas definir todas las dependencias necesarias. Esto se hace a través de un archivo de requisitos, lo que facilita la instalación automática durante la

construcción de la imagen.

```
requirements.txt:

# Ejemplo de dependencias para un agente BDI
# Sustituye por las librerías que uses (ej. para un agente en Python)
numpy==1.22.3
pandas==1.4.2
scikit-learn==1.0.2
matplotlib==3.5.2
tensorflow==2.8.0
scipy==1.8.0
```

2. Construcción de la Imagen Docker La construcción de la imagen se realiza mediante un archivo Dockerfile. Este archivo contiene las instrucciones para crear un entorno de ejecución, instalar las dependencias y copiar el código de tu agente.

```
Dockerfile:


# Usa una imagen base de Python oficial, que ya incluye el entorno de ejecución necesario
FROM python:3.9-slim

# Establece el directorio de trabajo dentro del contenedor
WORKDIR /app

# Copia los archivos de requisitos e instala las dependencias
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copia el código de tu aplicación de agente al directorio de trabajo
```

```
# 'bdi_agent/' es la carpeta donde está tu código fuente
COPY bdi_agent/ .

# Expone el puerto si tu agente necesita comunicarse (opcional)
# Por ejemplo, si el agente tiene una interfaz web o API REST
# EXPOSE 8080

# Comando para ejecutar el agente cuando el contenedor se inicie
# Asegúrate de que 'main.py' es el archivo de entrada de tu agente
CMD ["python", "main.py"]
```

Construcción y Ejecución del Contenedor

Una vez que tengas tu requirements.txt y tu Dockerfile en el mismo directorio que el código de tu agente (bdi_agent/), puedes construir la imagen y ejecutar el contenedor con los siguientes comandos:

Construir la Imagen

Ejecuta el siguiente comando en tu terminal. El punto (.) al final indica que Docker debe buscar el Dockerfile en el directorio actual.

```
#! sudo docker build -t asddosdetect .
```

docker build: Comando para construir una imagen.

-t mi-agente-bdi: Asigna un nombre (etiqueta o "tag") a la imagen. . El contexto de construcción, que es el directorio actual.

Ejecutar el Contenedor

Una vez que la imagen ha sido construida exitosamente, puedes ejecutar el agente dentro de un contenedor.

```
# ! sudo docker run --rm -it --privileged asddosdetect bash
```

docker run: Comando para crear y ejecutar un contenedor a partir de una imagen.

-d: Ejecuta el contenedor en modo detached (segundo plano). –name agente-en-ejecucion: Asigna un nombre al contenedor para poder gestionarlo fácilmente. mi-agente-bdi: El nombre de la imagen que creaste.

Comandos Adicionales Útiles: - Para ver los contenedores en ejecución: docker ps - Para ver los registros (logs) del agente: docker logs agente-en-ejecucion - Para detener el contenedor: docker stop agente-en-ejecucion - Para eliminar el contenedor (una vez detenido): docker rm agente-en-ejecucion

Esta metodología garantiza que tu agente BDI se pueda desplegar de forma predecible y consistente, sin importar el entorno de ejecución subyacente.

Resultados

```
2025-08-07 12:28:25,081 - __main__ - INFO - Modelo cargado exitosamente: ./models/ada_boost_training_exp_attack_tcp.jo
2025-08-07 12:28:25,081 - __main__ - INFO - Iniciando agente BDI de seguridad de red...
2025-08-07 12:28:25,083 - __main__ - INFO - Iniciando ciclo de monitoreo...
2025-08-07 12:28:25,083 - __main__ - INFO - Iniciando captura de tráfico...
2025-08-07 12:28:25,084 - __main__ - INFO - Ejecutando captura: tshark -i eth0 -a duration:30 -w pcap\07082025-122825.

=== AGENTE BDI DE SEGURIDAD DE RED ===
Comandos disponibles:
  status  - Ver estado del agente
```

```
  alerts  - Ver alertas recientes
  capture - Forzar captura manual
  quit    - Salir
====================================

>>> Capturing on 'Wi-Fi'
6112
2025-08-07 12:29:01,353 - __main__ - INFO - Procesando captura...
2025-08-07 12:29:02,403 - __main__ - INFO - Dataset cargado: output.csv, shape: (93, 25)
2025-08-07 12:29:02,404 - __main__ - INFO - Los atributos de las trazas de red son correctos
2025-08-07 12:29:02,415 - __main__ - INFO - Dataset reducido guardado: prod\output_reduced.csv
2025-08-07 12:29:02,463 - __main__ - INFO - (process_network_data) Procesando 93 registros de red
2025-08-07 12:29:02,498 - __main__ - INFO - Análisis completado: 1 ataques detectados de 93 flujos
```