sender.py - File Upload Manager

Overview

File: `client/sender.py`
Purpose: Monitor for new CSV files and upload them to the server
Role: Consumer in the producer-consumer architecture
Runs as: Standalone process (does NOT require admin privileges)

What It Does

`sender.py` is responsible for file monitoring and server communication:

1. **Monitors** `logs/pending_upload/` folder for new files every 2 seconds
2. **Detects** CSV files with `.ready` markers (signals file is complete)
3. **Reads** CSV and converts to JSON format
4. **Uploads** to server via HTTP POST
5. **Retries** failed uploads 3 times with exponential backoff
6. **Manages** files:
   - Success → Move to `logs/processed/`
   - Failure → Move to `logs/failed_uploads/`
7. **Auto-retries** failed uploads every 5 minutes

Architecture

```
logs/pending_upload/
  packets_20251129_220000.csv       ← Sniffer writes here
  packets_20251129_220000.csv.ready ← Ready marker
      ↓
   (sender.py detects)
      ↓
  Read CSV → Convert to JSON
      ↓
  HTTP POST to server
      ↓

  ↓ Success     ↓ Failure      ↓
Move to         Move to
processed/      failed_uploads/
```

Key Components

1. Configuration (Lines 18-33)

```
# Folder paths
PENDING_DIR = LOGS_DIR / 'pending_upload'    # Watch this folder
FAILED_DIR = LOGS_DIR / 'failed_uploads'     # Failed uploads
PROCESSED_DIR = LOGS_DIR / 'processed'       # Successful uploads

# Server configuration
SERVER_URL = "http://26.178.118.134:8000/ingest_packets"
POLL_INTERVAL = 2    # Check for new files every 2 seconds
MAX_RETRIES = 3      # Retry failed uploads 3 times
RETRY_DELAY = 5      # Wait 5 seconds before retry
BATCH_SIZE = 500     # Upload 500 packets per HTTP request
```

What you can change:

Variable	Default	Purpose	Effect of Changing
`SERVER_URL`	`http://...`	Server endpoint	Point to different server
`POLL_INTERVAL`	2 seconds	Check frequency	Lower = faster detection, higher CPU
`MAX_RETRIES`	3	Upload attempts	Higher = more resilient, slower failures
`RETRY_DELAY`	5 seconds	Delay between retries	Higher = less server load
`BATCH_SIZE`	500	Packets per request	Larger = fewer requests, bigger payloads

---

2. CSV Reading & Type Conversion (Lines 37-74)

Function: `read_csv_file(csv_path)`

Purpose: Read CSV and convert string values to proper Python types

The Problem: CSV files store everything as strings:

```
timestamp,tcp_syn,src_port
"1234567.123","True","80"
```

The Solution:

```
def read_csv_file(csv_path):
    # Read CSV
    with open(csv_path, 'r', encoding='utf-8') as f:
```

```
        reader = csv.DictReader(f)
        packets = list(reader)

    # Convert types
    for packet in packets:
        for key, value in packet.items():
            # Boolean conversion
            if value == 'True':
                packet[key] = True
            elif value == 'False':
                packet[key] = False

            # None conversion
            elif value == 'None' or value == '':
                packet[key] = None

            # Integer conversion for port numbers, etc.
            elif key in ['length', 'src_port', 'dst_port', ...]:
                packet[key] = int(value) if value else None

            # Float conversion for timestamp
            elif key == 'timestamp':
                packet[key] = float(value)
```

Result: Proper Python types for server API

---

3. Upload Logic (Lines 76-107)

Function: `upload_packets(packets, retry_count=0)`

Purpose: Send packet data to server via HTTP POST

```
def upload_packets(packets, retry_count=0):
    try:
        response = requests.post(
            SERVER_URL,
            json=packets,              # Auto-converts to JSON
            timeout=30,                # 30 second timeout
            headers={'Content-Type': 'application/json'}
        )

        if response.status_code == 200:
            logger.info(f"  Successfully uploaded {len(packets)} packets")
            return True
        else:
            logger.error(f"Server error: {response.status_code}")
```

```
            return False

    except requests.exceptions.ConnectionError:
        logger.error("Cannot connect to server")
        return False

    except requests.exceptions.Timeout:
        logger.error("Request timed out")
        return False
```

HTTP Request Example:

```
POST /ingest_packets HTTP/1.1
Host: 26.178.118.134:8000
Content-Type: application/json

[
  {
    "timestamp": 1701234567.123,
    "src_ip": "192.168.1.100",
    "dst_ip": "8.8.8.8",
    "protocol": "TCP",
    ...
  },
  {
    "timestamp": 1701234567.456,
    ...
  }
]
```

---

4. Retry Logic with Exponential Backoff (Lines 109-123)

Function: `upload_with_retry(packets)`

Purpose: Automatically retry failed uploads

```
def upload_with_retry(packets):
    for attempt in range(MAX_RETRIES):  # 3 attempts
        if upload_packets(packets, retry_count=attempt):
            return True  # Success!

        if attempt < MAX_RETRIES - 1:
            # Exponential backoff: 5s, 10s, 20s
            delay = RETRY_DELAY * (2 ** attempt)
            logger.info(f"Retrying in {delay} seconds...")
            time.sleep(delay)
```

```
    return False  # All retries failed
```

Retry Timeline:

```
Attempt 1: Upload fails
    ↓ Wait 5 seconds
Attempt 2: Upload fails
    ↓ Wait 10 seconds
Attempt 3: Upload fails
    ↓
Give up, move to failed_uploads/
```

Why exponential backoff? - Gives server time to recover - Reduces load on struggling server - Industry best practice

---

5. File Processing Pipeline (Lines 125-180)

Function: `process_csv_file(csv_path)`

Purpose: Complete workflow for one CSV file

Flow:

```
1. Check for .ready marker
   if not ready_marker.exists():
       return False  # Skip incomplete files
   ↓
2. Read CSV file
   packets = read_csv_file(csv_path)
   ↓
3. Upload in batches (if large file)
   for i in range(0, total_packets, BATCH_SIZE):
       batch = packets[i:i + BATCH_SIZE]
       upload_with_retry(batch)
   ↓
4. Move file based on result
   if success:
       csv_path.rename(PROCESSED_DIR / csv_path.name)
   else:
       csv_path.rename(FAILED_DIR / csv_path.name)
```

Why batching? - Large CSV might have 10,000+ packets - Uploading in one request = huge payload - Split into 500-packet batches = manageable

Example:

```
File with 2,000 packets:
  Batch 1: Packets 0-499    → Upload
```

```
Batch 2: Packets 500-999  → Upload
Batch 3: Packets 1000-1499 → Upload
Batch 4: Packets 1500-1999 → Upload
```

---

6. Failed Upload Retry (Lines 182-207)

Function: `retry_failed_uploads()`

Purpose: Periodically retry previously failed uploads

```python
def retry_failed_uploads():
    # Find all failed uploads
    failed_files = list(FAILED_DIR.glob('packets_*.csv'))

    if not failed_files:
        return  # Nothing to retry

    logger.info(f"Retrying {len(failed_files)} failed uploads...")

    for failed_file in failed_files:
        packets = read_csv_file(failed_file)

        if upload_with_retry(packets):
            # Success! Move to processed
            failed_file.rename(PROCESSED_DIR / failed_file.name)
            logger.info(f"  Retry successful")
        else:
            # Still failing, leave in failed_uploads/
            logger.warning(f"  Retry failed")
```

When called: - On startup: Retry all existing failed uploads - Every 5 minutes: Automatic retry

---

7. Main Monitoring Loop (Lines 209-247)

Function: `monitor_and_upload()`

Purpose: Continuously watch for new files

```python
def monitor_and_upload():
    last_retry_check = datetime.now()

    while True:  # Forever loop
        # 1. Find CSV files with .ready markers
        csv_files = []
        for ready_file in PENDING_DIR.glob('*.ready'):
```

```
            csv_file = ready_file.with_suffix('')  # Remove .ready
            if csv_file.exists() and csv_file.suffix == '.csv':
                csv_files.append(csv_file)

        # 2. Process in chronological order
        csv_files.sort()
        for csv_file in csv_files:
            process_csv_file(csv_file)

        # 3. Periodic retry of failed uploads (every 5 min)
        if (datetime.now() - last_retry_check).seconds > 300:
            retry_failed_uploads()
            last_retry_check = datetime.now()

        # 4. Wait before next check
        time.sleep(POLL_INTERVAL)  # 2 seconds
```

Timeline:

```
00:00 - Check for files → Process any found
00:02 - Check for files → None found
00:04 - Check for files → Process 1 file
00:05 - Retry failed uploads (5 min mark)
00:06 - Check for files → None found
...
```

Data Flow Example

Scenario: Uploading 1 CSV file with 150 packets

```
1. sniffer.py saves:
   logs/pending_upload/packets_20251129_220000.csv
   logs/pending_upload/packets_20251129_220000.csv.ready

2. sender.py (every 2 seconds):
   - Scans pending_upload/
   - Finds .ready marker
   - process_csv_file() called

3. Read CSV:
   packets = [
     {timestamp: 1701234567.123, src_ip: "192.168.1.1", ...},
     {timestamp: 1701234567.456, src_ip: "192.168.1.2", ...},
     ... (150 packets total)
   ]
```

```
4. Upload (no batching needed, < 500 packets):
   POST /ingest_packets
   JSON: [150 packets]

5. Server responds: 200 OK

6. Success! Move file:
   mv packets_20251129_220000.csv → logs/processed/
   rm packets_20251129_220000.csv.ready

7. Log: "  packets_20251129_220000.csv processed successfully (150 packets)"
```

Configuration Examples

Change Upload Frequency

```
POLL_INTERVAL = 5  # Check every 5 seconds instead of 2
```

Effect: Lower CPU usage, but slightly higher latency

Increase Retry Attempts

```
MAX_RETRIES = 5  # Try 5 times instead of 3
```

Effect: More resilient, but slower to fail

Reduce Batch Size (For Slower Networks)

```
BATCH_SIZE = 100  # Smaller batches
```

Effect: Smaller HTTP payloads, but more requests

Change Server URL

```
SERVER_URL = "http://192.168.1.10:8000/ingest_packets"
```

Effect: Upload to different server

Monitoring & Debugging

Check if Running

```
ps aux | grep sender.py
```

View Live Output

```
python sender.py
# Output shows:
# - Files being processed
# - Upload success/failure
# - Retry attempts
```

Check File Counts

```
# Pending uploads (waiting)
ls logs/pending_upload/*.csv | wc -l

# Processed (successful)
ls logs/processed/*.csv | wc -l

# Failed (need retry)
ls logs/failed_uploads/*.csv | wc -l
```

Tail Logs

```
# Watch sender.py output in real-time
python sender.py 2>&1 | tee sender.log
tail -f sender.log
```

Troubleshooting

No Files Being Uploaded

Checks: 1. Is sniffer.py running and creating files? `bash ls logs/pending_upload/`

2. Are `.ready` markers present?

   ```
   ls logs/pending_upload/*.ready
   ```

3. Can you reach the server?

   ```
   curl http://26.178.118.134:8000/health
   ```

Connection Errors

`ERROR: Connection error (attempt 1/3)`

Causes: - Server not running - Wrong SERVER_URL - Firewall blocking connection - Network issue

Solutions: 1. Check server is running: `bash curl http://SERVER_IP:8000/health`

2. Verify SERVER_URL in sender.py matches server
3. Check firewall allows port 8000

Files Stuck in failed_uploads/

Cause: Server consistently rejecting or timing out

Debug: 1. Manually upload one file: `python import requests packets = [...] # From CSV response = requests.post(SERVER_URL, json=packets) print(response.status_code, response.text)`

2. Check server logs for errors
3. Verify CSV format matches server expectations

High CPU Usage

Cause: POLL_INTERVAL too low

Solution:

```
POLL_INTERVAL = 5  # Check every 5 seconds
```

Duplicate Uploads

This should NOT happen due to atomic file moves

If it does: - Check for race conditions in sniffer.py - Ensure only one sender.py process running: `bash ps aux | grep sender.py # Should see only 1 process`

Performance Tips

For Large Files (>10,000 packets)

```
BATCH_SIZE = 1000  # Larger batches
```

For Slow Networks

```
BATCH_SIZE = 100   # Smaller batches
timeout=60         # Increase timeout in upload_packets()
```

For High-Latency Networks

```
RETRY_DELAY = 10    # Wait longer between retries
MAX_RETRIES = 5     # More attempts
```

File Organization

sender.py manages three directories:

pending_upload/

- Files waiting to be uploaded
- Created by sniffer.py
- Continuously monitored

processed/

- Successfully uploaded files
- Kept for audit trail
- Can be deleted after X days:

  ```
  find logs/processed -name "*.csv" -mtime +7 -delete
  ```

failed_uploads/

- Files that couldn't be uploaded
- Auto-retried every 5 minutes
- Manual retry: restart sender.py

Summary

sender.py is designed to: Continuously monitor for new files
Upload data reliably with retries
Handle network failures gracefully
Prevent data loss (move to processed/failed)
Auto-retry failed uploads
Batch large files for efficiency
Run independently from sniffer

It does NOT: Capture packets (that's sniffer.py's job)
Process or aggregate data (that's aggregator.py's job)
Run ML predictions (that's aggregator.py's job)

Next step: Ensure server (`main.py`) is running to receive uploads.