Done

- Added two new model configurations:
  - Simple Logistic Regression
  - Random Forest
  - SVM is too slow on this dataset (for now)
- Distributed framework implementation - Dask; powerful and easy.
- Change: Unbounded buffer
- Online benchmarking.

To do

- Build & Train SVM, Random Forest. Compare with current NN (Benchmark and document on offline performance)
- Adjust prediction and flow algorithms.
- Deploy lightweight & best model on testbed and profile performance.
  - Scaling performance
  - Collaboration in part is handled by the way that inferences are made on the local nodes and reported back to the master. Collection of evidence over time on **both** ends

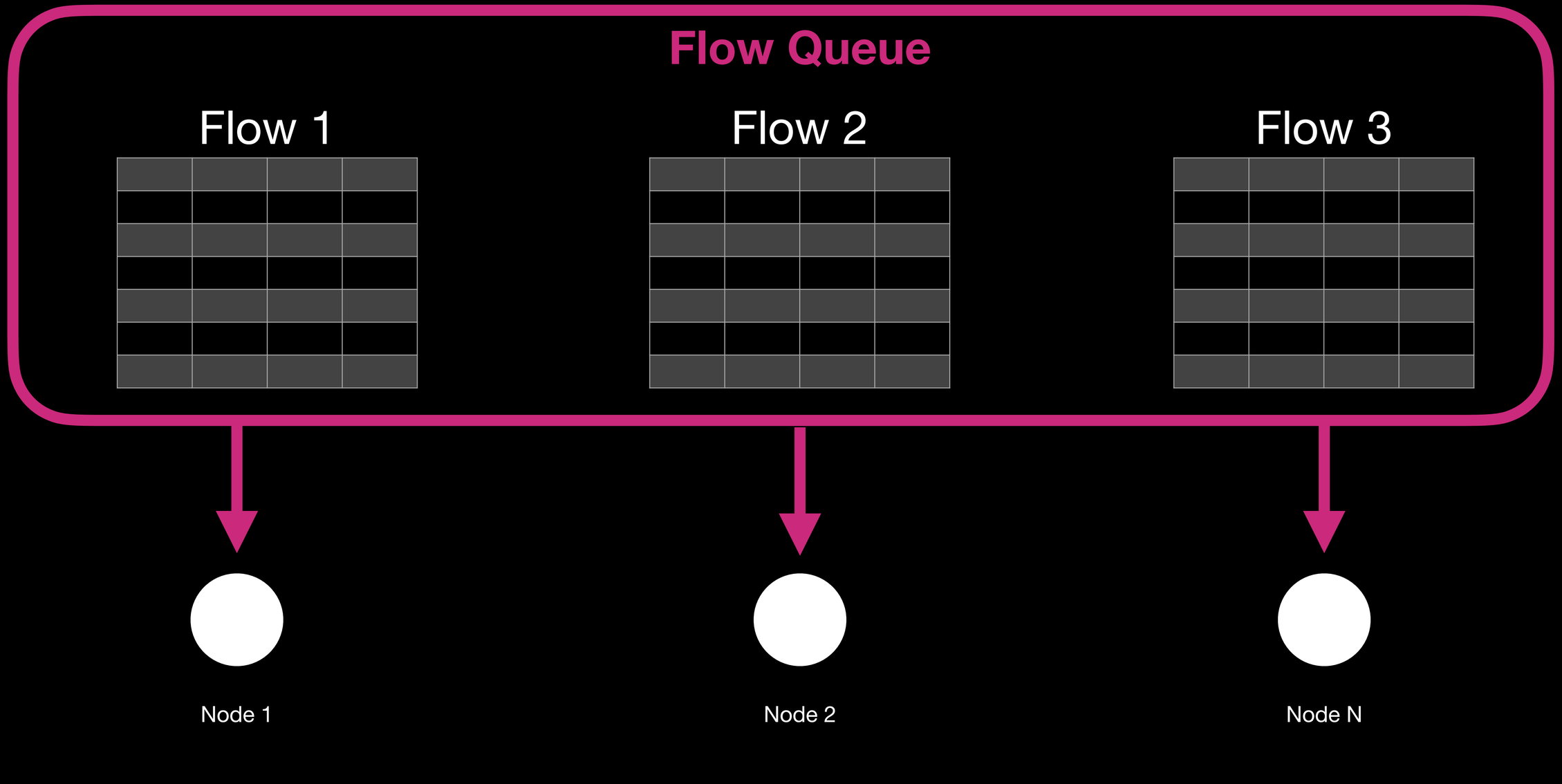
Distribution Scheme

- ***Raw Sockets -> Framework***
  - Much more condensed and managed
  - Transparent
  - Scalable
- ***Flow Generation:***
  - NFStream - automatically parallel
- DATA: 2,830,743 samples (total amongst all CSVs)

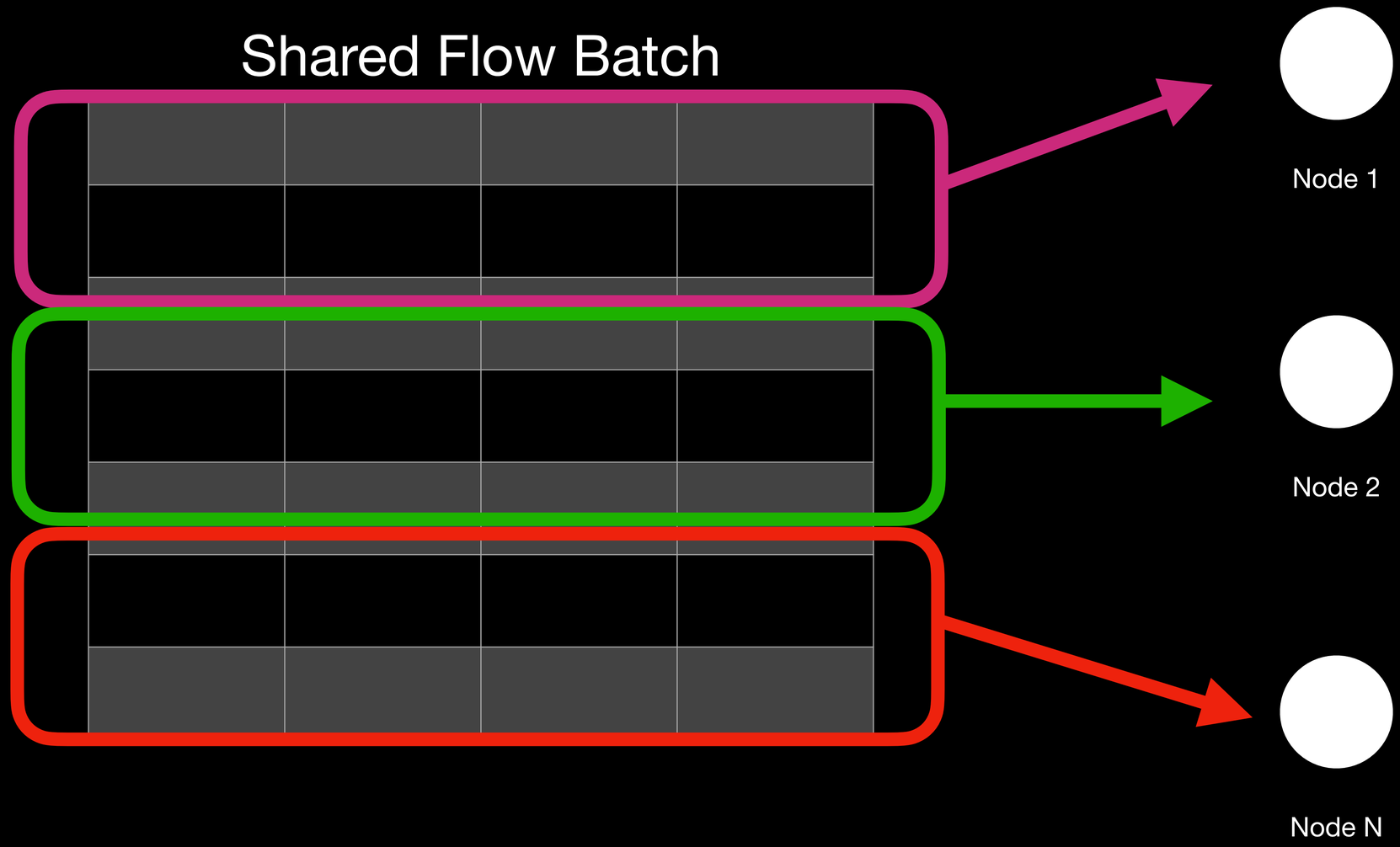
Data Delegation - Options

- **Option 1:**
  - Collect flows in increments and each node in the cluster gets handed a batch - our current approach.
- **Option 2:**
  - Collect flows into a larger batch, and have it distributed across the cluster with batch learning (available as an implicitly parallel operation in Ray).

Data Delegation - Option 1



Data Delegation - Option 2



Data Preprocessing

- CIC Dataset ~ 79 features
  - Our data is 46 **direct** features from NFStream. We reduce both to match.
  - Roughly half of the features are dropped, but we can always attempt to calculate them ourselves if need be.
- CIC Dataset is modified where all attacks are now marked “Malicious” instead of specific attack.
- In training, we change the Benign/Malicious to 0/1 respectively.

Learning Process

- Centralized training is slow on large datasets. Our CIC dataset is segmented into different days;
- This is perfect for federated learning whereby each node gets a piece of the data and locally trains on that data, then globally contributes to overall accuracy;
- At the end each node contains the fully trained model.
- This is more time-effective than centralized training.
- How does this fit into the pipeline?

Flow Processing

- Usually this will always be a bottleneck (aggregation time of statistics for each bidirectional communication of packets).
  - With NFStream, it is designed to be fast but it is still a slow process even if asynchronous to the server on realtime traffic.
  - Solution:
    - We capture packets into a temporary file (disk overhead), and still use NFStream to read and parse it due to its efficiency - use a large packet file to demonstrate its effectiveness.
    - Alternatively, if we had enough RAM, we could segment a small amount as RamDisk for the temporary file to reduce disk overhead.

Flow Benchmark - Offline

- Large Fuzzing capture file (493 MB - 2,244,139 total packets) for benchmarking.
  - Using NFStream to parse the pcap and our code to turn into a dataframe, it takes between 6 to 9 seconds - depending on the system.

```
Starting stream read...
File "Fuzzing_pcap.pcapng" size: 470.12377548217773 MB
Time to read & convert to dataframe using NFStream: 6.700629949569702 seconds

      Destination Port  Flow Duration  Total Fwd Packets  ...  URG Flag Count  CWE Flag Count  ECE Flag Count
0               1900.0          189.0                6.0  ...             0.0             0.0             0.0
1               1900.0          207.0                6.0  ...             0.0             0.0             0.0
2               1900.0          206.0                6.0  ...             0.0             0.0             0.0
3               1900.0          202.0                6.0  ...             0.0             0.0             0.0
4              51002.0            0.0                1.0  ...             0.0             0.0             0.0
...                ...            ...                ...  ...             ...             ...             ...
1802           51002.0            0.0                1.0  ...             0.0             0.0             0.0
1803           51002.0            0.0                1.0  ...             0.0             0.0             0.0
1804           51002.0            0.0                1.0  ...             0.0             0.0             0.0
1805           51002.0            0.0                1.0  ...             0.0             0.0             0.0
1806           51002.0            0.0                1.0  ...             0.0             0.0             0.0

[1807 rows x 46 columns]
```

Model Architectures

- **Intermediate Complexity Binary Classifier NN (PyTorch):** ~93% Testing accuracy; Much more aggressive model in its malicious detections.
- **Logistic Regression (Scikit):** ~81.873877250802 72% Testing accuracy; Medium aggressiveness.
- **Random Forest (Scikit):** ~99.769438590039 19% Testing accuracy; Lenient model.
- **LSTM?**

Learning Benchmark - Offline

- Complete benign capture (new random capture).

source	likely benign flows	likely malicious flows	final judgment
10.10.0.100	18	1	likely benign node
10.10.0.139	1	1	likely benign node
10.10.0.226	1	0	likely benign node
10.10.0.33	1	0	likely benign node
fe80::10d4:b134:d4f0:2b79	1	0	likely benign node
fe80::fad0:27ff:fe8a:d27b	1	0	likely benign node
10.10.0.131	1	0	likely benign node

- Our first attack captures (ssh brute force).

source	likely benign flows	likely malicious flows	final judgment
192.168.0.122	184	1805	likely malicious node
192.168.0.104	288	673	likely malicious node
192.168.0.41	0	6	likely malicious node
192.168.0.1	10	1	likely benign node
fe80::ba27:ebff:fe2e:597d	8	5	likely benign node
fe80::dea6:32ff:fe6e:363	7	6	likely benign node
fe80::dea6:32ff:fe8a:e7fe	3	3	likely benign node
fe80::e65f:1ff:fe85:5d50	3	4	likely malicious node
192.168.0.176	10	1	likely benign node
192.168.0.22	0	28	likely malicious node
192.168.0.21	1	6	likely malicious node
192.168.0.214	1	0	likely benign node
192.168.0.109	10	1	likely benign node
fe80::562a:1bff:fecc:583a	1	0	likely benign node
fe80::4aa6:b8ff:feff:a63a	1	0	likely benign node

From Zero (offline) Deployment Pipeline

- Step 1: Train all known nodes in a **federated** manner.
  - 1.5: Consider “best” model; e.g., accuracy, performance.
- Step 2: Begin IDS System
  - Consider data delegation strategy (collaboration).
- Step 3: Scaling
  - New nodes can join asynchronously and be served the model - already trained in step 1.
  - Now these nodes obtain their own flow batches and can predict using the model.
  - Many nodes can cause many aggregation requests (master node needs to be reasonably resource-capable).

Deployment Architectures

1. **Segmented strategy (first topology)**
    - IDS system and access points/routers are entirely separate. Scripts and requirements can be provided individually as part of the system
2. **Unified strategy (secondary topology)**
    - IDS system and access points/routers are all unified. The technologies and requirements would be provided as part of a router’s OS out-of-box.

Model Distribution Technique

- Nodes joining is analogous to creating a copy of the memory space of a parent process. Thus any new node, is instantiated a new model object based on the subtype we specify.
- Advantage: Easily switch between Pytorch, Scikit, or others at runtime using a single line of code and not have to change the behavior of the core compute function.

Proposed Collaborative Inferenecing - #1

- **Compute Nodes -** Receive data frame and perform a one-to-one prediction of flows (malicious or benign).
  - Collection of “Evidence” over time for sources. After a threshold is reached, we can report an (un)official judgment from that node.
  - These nodes then flush their prediction buffer to prevent memory abuse.
- **Master Node -** As the judgements come in from the nodes, we can build our own evidence buffer and repeat the process on the master node after the master’s threshold is reached; then flush.
- **Dubbing the “Jury-Judge model”.**