Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta

Logistics

- HW5 and MP3 due today.

- Final exam period May 7 – May 15.
  - Syllabus includes everything covered up to and including distributed datastores.
  - Relative weightage:
    - Midterm 1 and Midterm 2 contents: 55-60%
    - Post-midterm2 contents: 40-45%

Logistics

- Next Wednesday's class:
  - No new materials will be covered.
  - Quick reminder of topics covered in class.
  - Open-ended Q/A on topics of your choice.

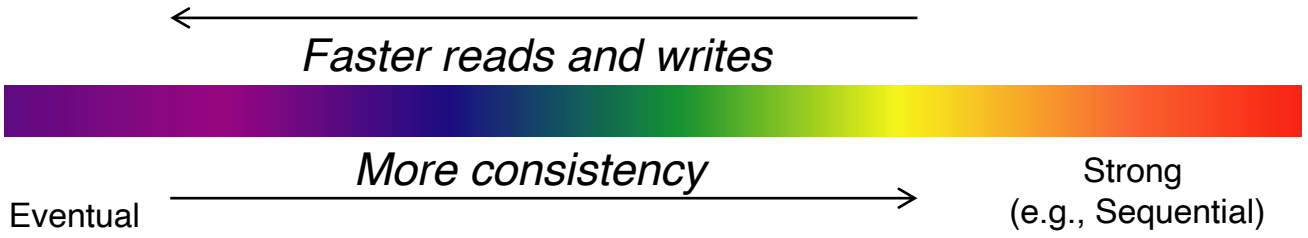
- May 6th is last day of instruction: no OHs after that.
  - Come prepared with your questions on Wednesday.
  - We will continue monitoring Campuswire until final exams end.

Today's focus

- Brief overview of key-value stores
- Distributed Hash Tables
  - Peer-to-peer protocol for efficient insertion and retrieval of key-value pairs.
- Key-value stores in the cloud
  - How to run large-scale distributed computations over key-value stores?
    - Map-Reduce Programming Abstraction
    - Cloud Scheduling
  - How to design a large-scale distributed key-value store?
    - Case-study: Facebook's Cassandra

Eventual Consistency

- Cassandra offers Eventual Consistency
  - If writes to a key stop, all replicas of key will converge.
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems



Read Quorums

- Reads
  - Client specifies value of R (≤ N = total number of replicas of that key).
  - R = read consistency level.
  - Coordinator waits for R replicas to respond before sending result to client.
  - In background, coordinator checks for consistency of remaining (N-R) replicas, and initiates read repair if needed.

Write Quorums

- Client specifies W (≤ N)
- W = write consistency level.
- Client writes new value to W replicas and returns when it hears back from all.
  - Default strategy.

Quorums in Detail (Contd.)

- R = read replica count, W = write replica count
- Necessary conditions for consistency:
  1. W+R > N
     - Write and read intersect at a replica. Read returns latest write.
  2. W > N/2
     - Two conflicting writes on a data item don't occur at the same time.
- Select values based on application
  - (W=N, R=1):
    - great for read-heavy workloads
  - (W=1, R=N):
    - great for write-heavy workloads with no conflicting writes.
  - (W=N/2+1, R=N/2+1):
    - great for write-heavy workloads with potential for write conflicts.
  - (W=1, R=1):
    - very few writes and reads / high availability requirement.

Cassandra Consistency Levels

- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator may cache write and reply quickly to client
  - ALL: all replicas
    - Slowest, but ensures strong consistency
  - ONE: at least one replica
    - Faster than ALL, and ensures durability without failures
  - QUORUM: quorum across all replicas in all datacenters (DCs)
    - Global consistency, but still fast
  - EACH_QUORUM: quorum in every DC
    - Lets each DC do its own quorum: supports hierarchical replies
  - LOCAL_QUORUM: quorum in coordinator's DC
    - Faster: only waits for quorum in first DC client contacts

Eventual Consistency

- Sources of inconsistency:
  - Quorum condition not satisfied R + W < N.
    - R and W are chosen as such.
    - when write returns before W replicas respond.
      - Sloppy quorum: when value stored elsewhere if intended replica is down, and later moved to the replica when it is up again.
  - When local quorum is chosen instead of global quorum.
- Hinted-handoff and read repair help in achieving *eventual consistency*.
  - If all writes (to a key) stop, then all its values (replicas) will converge eventually.
  - May still return stale values to clients (e.g., if many back-to-back writes).
  - But works well when there a few periods of low writes – system converges quickly.

Cassandra vs. RDBMS

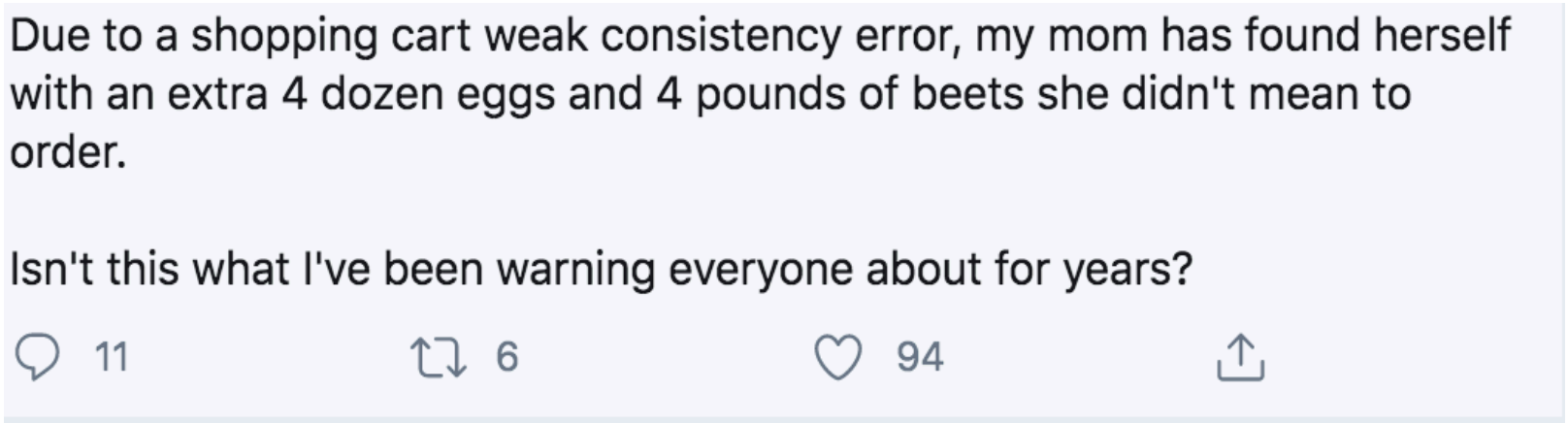
- MySQL is one of the most popular RDBMS (and has been for a while)
- On > 50 GB data
- MySQL
  - Writes 300 ms avg
  - Reads 350 ms avg
- Cassandra
  - Writes 0.12 ms avg
  - Reads 15 ms avg
- Orders of magnitude faster.

Other similar NoSQL stores

- Amazon's DynamoDB
  - Cassandra's data partitioning, replication, and eventual consistency strategies inspired from Dynamo.
  - Uses sloppy quorum as the default mechanism for eventual consistency with availability.
  - Uses vector clocks to capture causality between different versions of an object.
  - Dynamo: Amazon's Highly Available Key-value Store, SOSP'2007.
- LinkedIn's Voldemort
  - Inspired from DynamoDB.
- …..

Is it a good idea to trade-off consistency for availability?

A tweet by a distributed systems researcher:



Summary

- CAP theorem: cannot only achieve 2 out of 3 among consistency, availability, and partition-tolerance.

- Partition-tolerance is required in distributed datastores.
  - Choose between consistency and availability.

- Many modern distributed NoSQL key-value stores (e.g. Cassandra) choose availability, providing only eventual consistency.

Other examples of distributed systems

Distributed Machine Learning Training

- Training split across multiple GPUs (across multiple servers).
- Two forms of parallelism:
  - Data parallelism: too much data
    - each GPU operates on different shards of data.
    - each GPU holds full copy of the model.
  - Model parallelism: too big model.
    - model is split across GPUs, each GPU works on same data.
    - Pipeline parallelism: model too deep (too many layers)
    - Tensor parallelism: layer too wide.
- In practice today: hybrid – data, tensor, and pipeline parallelism.
  - 3D parallelism
  - Recent frameworks: Megatron (NVIDIA), Deepspeed (Microsoft)

Distributed Machine Learning Training

- Communication Collectives:
  - inter-GPU communication abstraction
  - been around for a while:
    - part of MPI (message passing interface) standard

Distributed Machine Learning Training

- Examples of collectives:

1. Broadcast

One process sends the **same data** to all others.

```
Before:   P0: [A]   P1: [_]   P2: [_]   P3: [_]
After:    P0: [A]   P1: [A]   P2: [A]   P3: [A]
```

Used for: Syncing initial model weights, sharing hyperparameters, broadcasting batch metadata.

Distributed Machine Learning Training

- Examples of collectives:

2. Scatter

One process sends **different chunks** to each process.

```
Before:   P0: [A|B|C|D]   P1: [_]   P2: [_]   P3: [_]
After:    P0: [A]          P1: [B]   P2: [C]   P3: [D]
```

Used for: Distributing dataset shards, sending different micro-batches to pipeline stages.

Distributed Machine Learning Training

- Examples of collectives:

3. Gather

One process **collects chunks** from all others — inverse of scatter.

```
Before:   P0: [A]    P1: [B]    P2: [C]    P3: [D]
After:    P0: [A|B|C|D]
```

Used for: Collecting predictions, assembling sharded outputs on a root process.

Distributed Machine Learning Training

- Examples of collectives:

4. All-Gather

Every process ends up with **all chunks** from all processes.

```
Before:   P0: [A]    P1: [B]    P2: [C]    P3: [D]
After:    P0: [A|B|C|D]    P1: [A|B|C|D]    P2: [A|B|C|D]    P3: [A|B|C|D]
```

Used for: ZeRO-3 parameter reconstruction before forward pass, Megatron tensor parallel weight gathering.

Distributed Machine Learning Training

- Examples of collectives:

5. Reduce

One process collects data from all and **applies a reduction** (sum, max, avg).

```
Before:   P0: [A]    P1: [B]    P2: [C]    P3: [D]
After:    P0: [A+B+C+D]
```

Used for: Collecting loss values to a master process, aggregating metrics.

Distributed Machine Learning Training

- Examples of collectives:

6. All-Reduce

Every process gets the **reduced result** — most important collective in deep learning.

```
Before:   P0: [A]    P1: [B]    P2: [C]    P3: [D]
After:    P0: [A+B+C+D]    P1: [A+B+C+D]    P2: [A+B+C+D]    P3: [A+B+C+D]
```

Used for: Gradient synchronization in data parallelism (DDP), Megatron tensor parallel output combination.

Distributed Machine Learning Training

- Examples of collectives:

7. Reduce-Scatter

Every process contributes data; result is **reduced and scattered** so each process gets one chunk.

```
Before:   P0: [A0|A1|A2|A3]
          P1: [B0|B1|B2|B3]
          P2: [C0|C1|C2|C3]
          P3: [D0|D1|D2|D3]

After:    P0: [A0+B0+C0+D0]
          P1: [A1+B1+C1+D1]
          P2: [A2+B2+C2+D2]
          P3: [A3+B3+C3+D3]
```

Used for: ZeRO gradient sharding — each GPU only keeps the gradient shard it owns.

Key insight: All-Reduce = Reduce-Scatter + All-Gather

Distributed Machine Learning Training

- Realizing a collective operation: Tree AllReduce

6. All-Reduce

Every process gets the **reduced result** — most important collective in deep learning.

```
Before:   P0: [A]    P1: [B]    P2: [C]    P3: [D]
After:    P0: [A+B+C+D]    P1: [A+B+C+D]    P2: [A+B+C+D]    P3: [A+B+C+D]
```

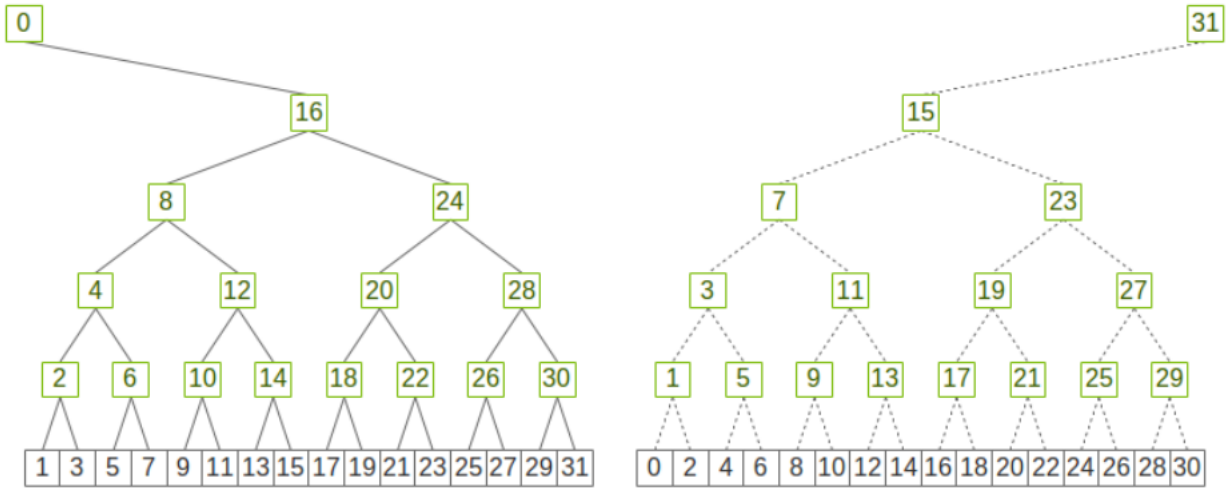


image source: NCCL's website

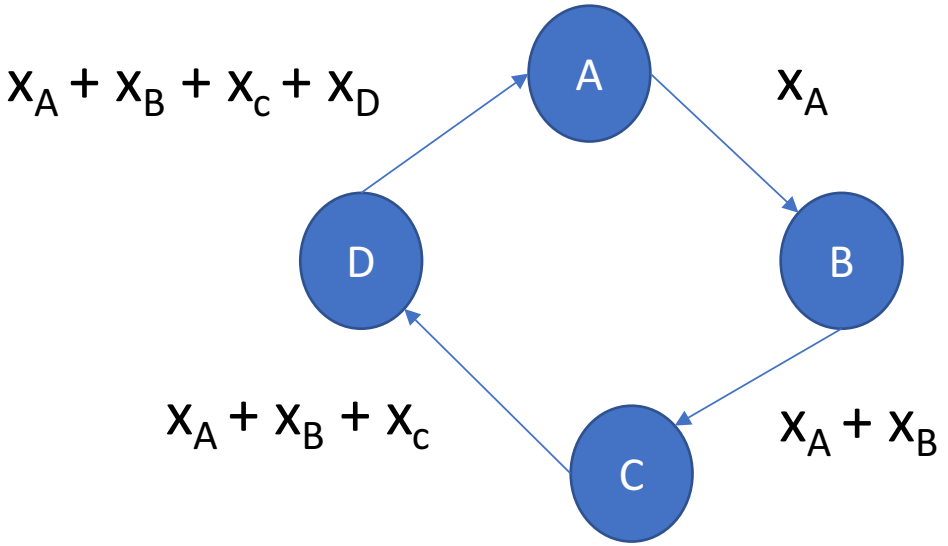
Distributed Machine Learning Training

- Realizing a collective operation: Ring AllReduce

6. All-Reduce

Every process gets the **reduced result** — most important collective in deep learning.

```
Before:   P0: [A]    P1: [B]    P2: [C]    P3: [D]
After:    P0: [A+B+C+D]    P1: [A+B+C+D]    P2: [A+B+C+D]    P3: [A+B+C+D]
```



Distributed Machine Learning Training

- Research questions:
  - How to make models / training frameworks more light-weight?
  - How to split the model? How to pipeline model?
  - How to schedule training jobs on available GPUs?
  - How to reduce size of network transfers?
  - How to speed up intra-server GPU communication?
  - How to speed up inter-server GPU communication?
  - …….

- Rich, but highly crowded, space.

Federated Learning

- Key goal: data privacy

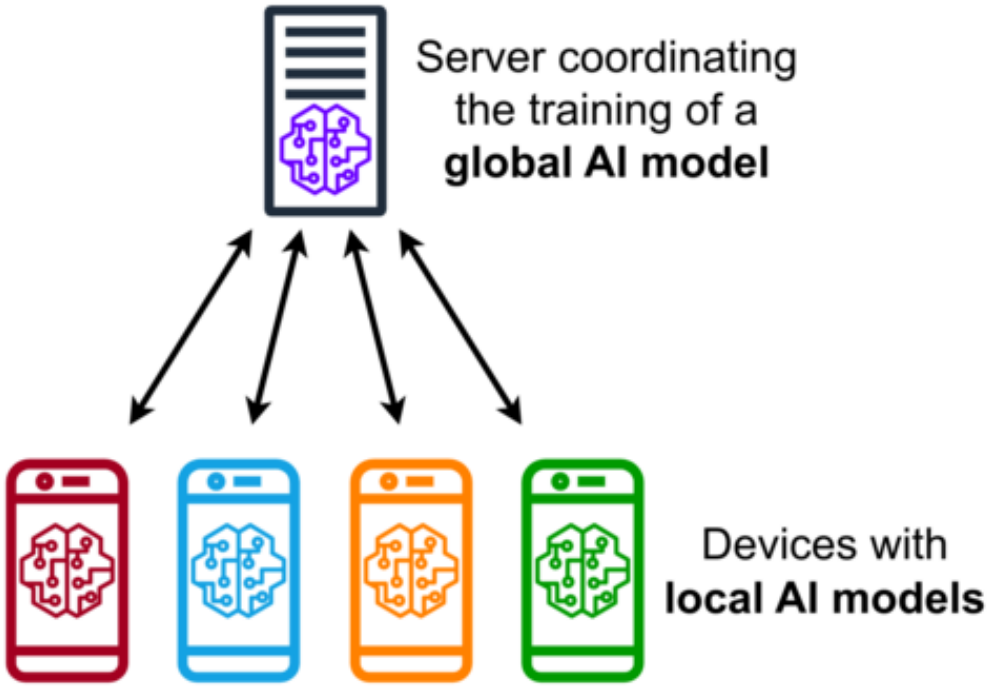
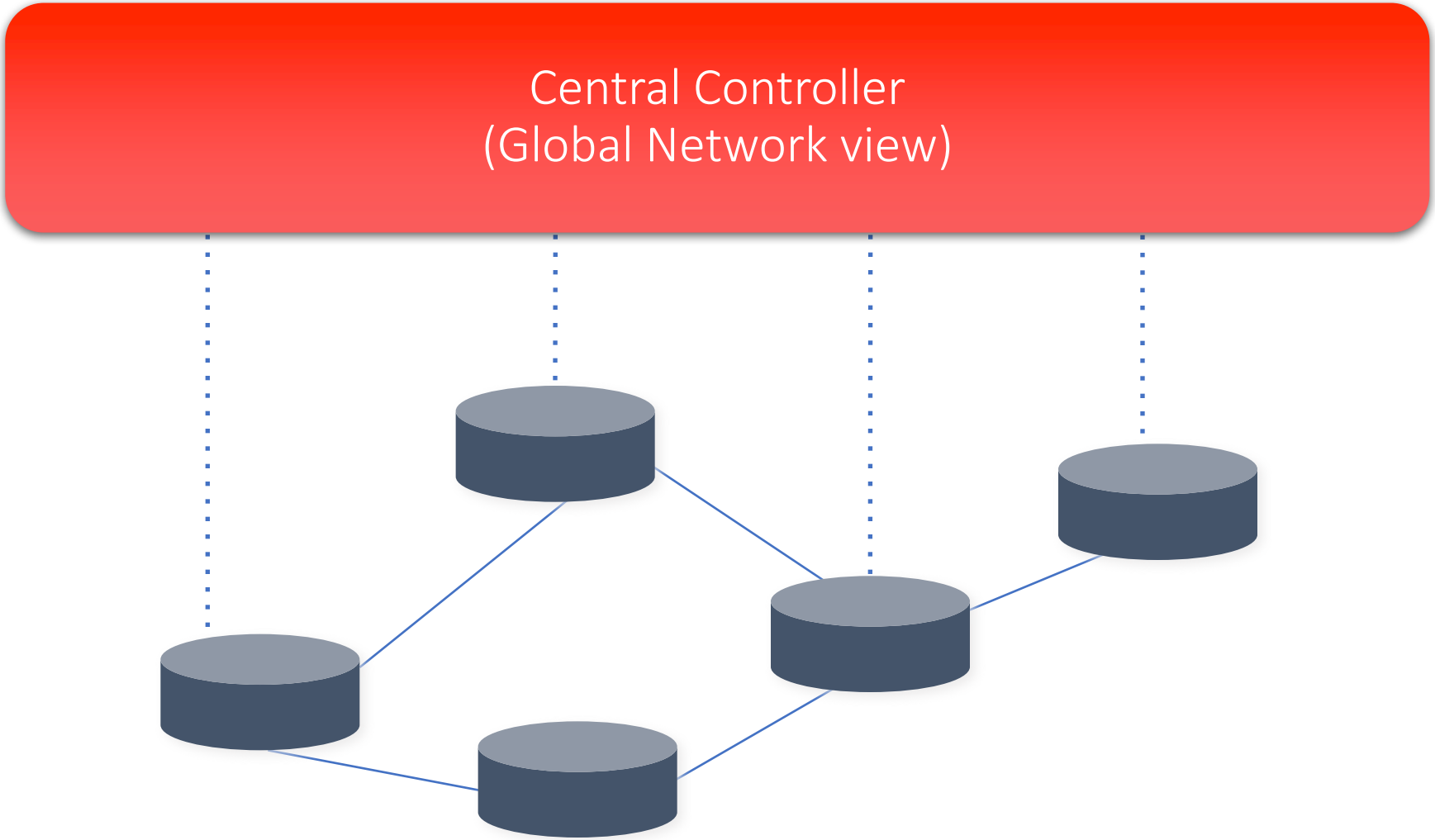


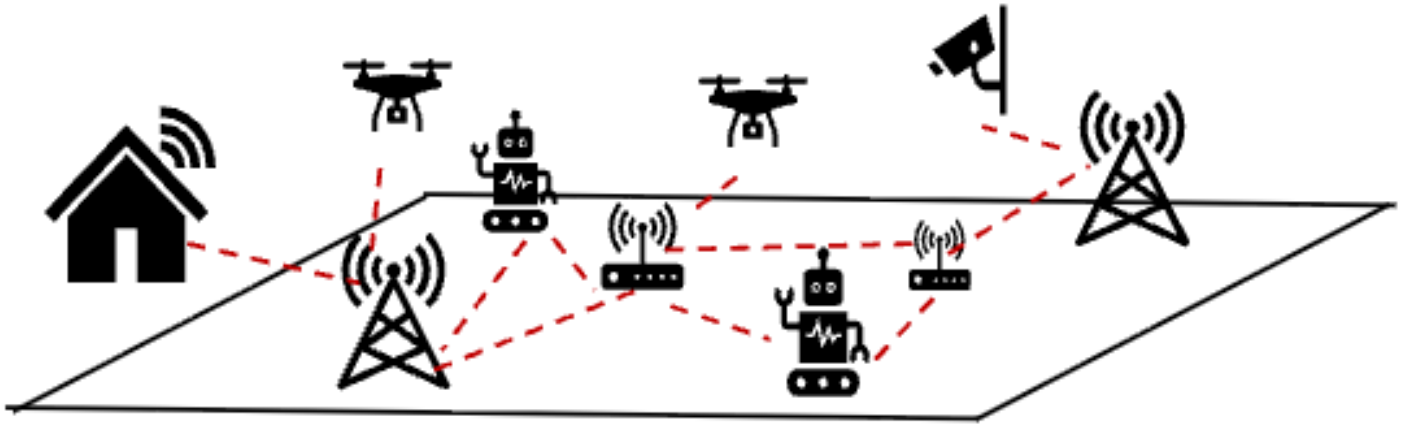
image source: wikipedia

Software-Defined Networking



Internet of Things (IoT)

Example: autonomous farms, home automation, industrial IoT, etc.



Multi-cell Deployments

Thank you!

go.illinois.edu/flex