0.1 Distributed Systems are Trees on Top of DAGs on Top of Graphs

From ./AE-Specifications-ETH/standalone/Trees-DAGs-Graphs.tex

This essay explores the layered graph-theoretic nature of distributed systems. At the lowest layer, physical and logical interconnects form undirected **graphs**. On top of this lie **DAGs** representing dependency, scheduling, and locking relationships. At the top, application-level consistency and authority are imposed via **trees** such as namespace hierarchies, leadership structures, and commit chains. We further examine how modern datacenters, populated by diverse xPUs (CPUs, GPUs, IPUs, DPUs), break the illusion of shared memory and necessitate protocol designs that exploit the native graph structure using mechanisms such as RDMA.

0.1.1 Graphs: The Physical and Logical Fabric

The physical topology of a datacenter is a graph: nodes represent compute units (CPUs, GPUs, IPUs, etc.) and edges represent communication links (Ethernet, NVLink, InfiniBand, etc.). These links may have diverse properties:

- Bandwidth and latency asymmetries
- Failures or congestion under load
- Scheduled or dynamic routing paths

Unlike the shared memory abstraction, these links form a non-uniform, fault-prone, and inherently asynchronous substrate. Real computation in modern datacenters occurs *on this graph*—not above it.

0.1.2 DAGs: Causality and Locking

On top of the physical graph lies a directed acyclic graph (DAG) representing **causality, scheduling, and consistency constraints**. DAGs arise in:

- **Transaction dependencies**: Operations must follow a directed order to preserve causality.
- **Lock hierarchies**: Preventing deadlock requires acquiring locks in a fixed topological order.
- **Build systems and job schedulers**: Tasks must respect dependencies.

Locking as a DAG

Databases employ lock hierarchies structured as DAGs to prevent circular waits. For example, the following might form a hierarchy:

1. Lock table
2. Then row

3. Then field

Each level narrows scope and follows a partial order. Enforcing that locks are acquired in topological order avoids cycles and hence deadlock.

0.1.3 Trees: Names, Commit Chains, and Leaders

At the top of the stack are trees. These structures are usually logical:
- **Namespace hierarchies**: e.g., file systems, DNS.
- **Leadership trees**: elected leaders per region, rack, or quorum.
- **Consensus and commits**: commit chains or logs form trees (or more precisely, forests with fork resolution).

These trees impose structure on the otherwise messy DAGs and graphs below, enabling:
- Easier authority delegation
- Fault domain containment
- Clear lineage and rollback support

0.1.4 Breaking the Shared Memory Illusion

Shared memory simplifies programming but breaks down in distributed xPU environments:
- Memory isn't uniformly addressable
- Coherence protocols are expensive or infeasible
- Latency variance introduces uncertainty in synchronization

RDMA: Network as Memory Bus

Remote Direct Memory Access (RDMA) partially restores shared memory semantics:
- Allows direct writes/reads between NICs with low latency
- Bypasses kernel and CPU involvement
- Supports zero-copy semantics for performance

But RDMA also forces a shift:
- You must think **asynchronously**
- Buffers must be explicitly registered and tracked
- Failures are explicit, not hidden

0.1.5 Exploiting the Graph: The Path Forward

To fully exploit xPU networks:
- Treat communication paths as first-class citizens
- Build coordination mechanisms that reflect graph topology
- Favor protocols that can adapt dynamically to congestion and partitioning

New system designs should:

1. Replace locking with message-passing wherever feasible

2. Encode application semantics in DAGs, not linear logs
3. Use explicit versioning and conflict resolution mechanisms

0.1.6 Conclusion

Distributed systems are not built on the abstraction of shared memory. They are constructed on a layered composition:

Graphs: physical connectivity

DAGs: causal and logical dependencies

Trees: naming, consensus, and leadership

The challenge of distributed systems is to harmonize these layers while respecting the physical realities of the system. To do so, we must leave behind illusions of synchrony and embrace graph-native programming models.