Contents

• Monday 16.2.
  • Dynamo and NoSQL
  • SDN and Clouds

• Thursday 19.2.
  • Advanced topics and summary
Amazon Dynamo

Aim is to store various kinds of data and have high availability

Build a distributed storage system that scales

Simple: **key-value**
- Highly available
- Guarantee **Service Level Agreements (SLA)**

Based on the SOSP 2007 presentation and paper: Dynamo: Amazon’s Highly Available Key-value Store
System Assumptions and Requirements

Query Model: simple read and write operations to a data item that is uniquely identified by a key

ACID Properties: Atomicity, Consistency, Isolation, Durability

Consistency is relaxed for efficient operation → noSQL data stores such as Dynamo

Efficiency: latency requirements which are in general measured at the 99.9th percentile of the distribution

Other Assumptions: operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization
Service Level Agreements (SLA)

Application can deliver its functionality in **bounded time**: Every dependency in the platform needs to deliver its functionality with even tighter bounds.

Multi-hop routing increases variability in routing. Dynamo is a one-hop DHT.

Example: *service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second*
Dynamo Design Consideration

Sacrifice strong **consistency** for **availability**

Conflict resolution is executed during **read** instead of **write**
  Use quorums and other techniques

Other principles:
  Incremental scalability
  Symmetry
  Decentralization
  Heterogeneity
Inconsistencies in distributed systems

Network partitions

Node is down

Process crashes before flushing

File corruption

...
CAP Theorem

CAP, first conceived in 2000 by Eric Brewer and formalized into a theorem in 2002 by Nancy Lynch

The proof involves a scenario in which a replicated service receives two conflicting requests when the processing sites are partitioned leading to at least one inconsistent response compared to a one-copy semantics.

A useful model for describing the fundamental behavior of NoSQL systems

CAP is generally described as following:
Of three desirable properties you want in your system: **consistency, availability** and **tolerance** of network partitions,

*you can only choose two.*
## Summary of Techniques in Dynamo

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff (use another server for replica if proper one is not available)</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available.</td>
</tr>
<tr>
<td>Recovering from permanent failures</td>
<td>Anti-entropy using Merkle trees (summarization of key ranges of virtual nodes)</td>
<td>Synchronizes divergent replicas in the background.</td>
</tr>
<tr>
<td>Membership and failure detection</td>
<td>Gossip-based membership protocol and failure detection.</td>
<td>Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.</td>
</tr>
</tbody>
</table>
Dynamo Implementation

Data Stores
Nodes in the system are spread around a logical circle
Nodes are responsible for the region between it and its predecessor
**Virtual nodes** are evenly dispersed and appear to be regular nodes in the system, but in reality are just handled by the nodes of the system
Can be geographically distributed

Object Data
Uses hashing of an object’s key to determine where to store the object
Each object is replicated across N nodes (N-1 successor nodes to the coordinator node)
Consistent Hashing Revisited

Properties

**Smoothness** → addition of bucket does not cause movement between existing buckets

**Spread & Load** → small set of buckets that lie near object

**Balance** → no bucket is responsible for large number of objects

Moderate load imbalance is possible

Virtual nodes address this

**Log n replication** factor gives $O(\text{items} / n)$ balance with high probability for a high number of uniformly distributed items
Partition Algorithm

Consistent hashing: the output range of a hash function is treated as a fixed circular space or “ring”.

“Virtual Nodes”: Each node can be responsible for more than one virtual node.

Virtual nodes are needed to address data/node imbalance problem.
Replication

Each data item is replicated at N hosts

“preference list”: The list of nodes that is responsible for storing a particular key (physical nodes)

Multiple data centers:
Preference list is configured so that each object is replicated across multiple data centers
Data Versioning

A `put()` call may return to its caller before the update has been applied at all the replicas.

A `get()` call may return many versions of the same object.

**Challenge:** an object having distinct version subhistories, which the system will need to reconcile in the future.

**Solution:** uses **vector clocks** in order to capture causality between different versions of the same object.
A vector clock is a list of \textit{(node, counter)} pairs.

Every version of every object is associated with one vector clock.

If the counters on the first object’s clock are less-than-or-equal to all of the counters in the second clock, then the first is an ancestor of the second and can be forgotten.

Figure from the Dynamo article.
**Sloppy Quorum**

The sloppy quorum technique is used to handle temporal faults.

Read/Write involves \( N \) nodes (preference list) \( R,W \) are the minimum number of nodes that must participate in a successful read/write operation.

Setting \( R + W > N \) yields a quorum-like system.

In this model, the latency of a get (or put) operation is dictated by the slowest of the \( R \) (or \( W \)) replicas.

\( R \) and \( W \) are usually configured to be less than \( N \), to provide better latency.

Typical values \((3,2,2) \) \((N,R,W)\)
Tuning R and W

With 5 nodes can set R=W=5 for high consistency, but now the solution is vulnerable to network partitions. This also burdens the cluster heavily.

Can set R=W=1, but this may result in weak consistency with failures.

Default values (3,2,2) balance between the performance and consistency: two nodes should respond when reading/writing out of the three replicas.
Dynamo Execution

Writes
Requires generation of a new vector clock by coordinator
Coordinator writes locally
Forwards to N-1 nodes, if W-1 respond then the write was successful

Reads
Forwards to N-1 nodes, if R-1 respond then forwards to user
Only unique responses forwarded
User handles merging if multiple versions exist
User has to write back the definite answer
Gossip

A **gossip**-based protocol propagates membership changes and maintains an **eventually consistent** view of membership.

Each node contacts a peer chosen at **random every second**.

The two nodes efficiently reconcile their persisted membership change histories.

Also reconcile position information on the ring (virtual buckets).
Hinted handoff

The hinted handoff is also used to handle temporal faults

Assume $N = 3$. When $A$ is temporarily down or unreachable during a write, send replica to $D$

$D$ is hinted that the replica belongs to $A$ and it will deliver to $A$ when $A$ is recovered

As a result $A$ is always writable
Eventual consistency

Given the **CAP impossibility result**, weaker consistency models were proposed to ensure availability and high performance.

**Eventual consistency**

All servers converge to the same state, but can be in differing states while converging.

A weak property

Works with network partitions

If a network partition happens the updates are eventually propagated to all servers.
Anti-entropy overview

To ensure eventual consistency, replicas exchange information about the data (writes) they have witnessed.

This information exchange is called **anti-entropy** 
*after reversing entropy (thermodynamic randomness)*

Multiple ways to implement anti-entropy

- Should be an asynchronous process
- Broadcast, multicast, …
- A replica typically broadcasts a write to other replicas
- A logical clock is used to handle concurrent writes
Anti-entropy (replica synchronization)

A Merkle tree is a hash tree where leaves are hashes of the individual keys.

Can check each branch independently without downloading the full data.

Used to verify consistency of replicas (which replicas are out of sync).

One Merkle tree for a virtual node (key range).

Make divergent data sets consistent.
Merkle Tree

Block B1
Block B2
Block B3
Block B4
Block B5
Some benefits of anti-entropy

No need for complex failure code or master node election

Writes are fast, only local state is updated before returning

The replicas are updated with an asynchronous background process

Replicas can also be located in more distant data centers
Anti-entropy and guarantees

Typical requirements:
  - **Safety**: nothing bad ever happens
  - **Liveness**: something good eventually happens

Anti-entropy does not guarantee safety

But works reasonably well in practice

There are tools to assess the consistency of an anti-entropy system
  Determine inconsistency window
  and then set the parameters
CALM theorem

CALM means *consistency as logical monotonicity*

Informally, programs that compute an ever-growing set of facts and do not retract emitted facts can be run safely on an eventually consistent store

Example:
- Stock trades $\rightarrow$ CALM
- Latest stock trade $\rightarrow$ Not CALM

Reading:
*Eventual Consistency Today: Limitations, Extensions, and Beyond* by Peter Bailis, Ali Ghodsi
http://queue.acm.org/detail.cfm?id=2462076
On Replication and Sharding

A distributed system may use the following techniques:

Sharding that distributes different data items across multiple servers, each server is a single source.

Replication copies data across servers and data items can be found on multiple servers.

Replication is either master-slave or peer-to-peer.

Master maintains authoritative version for writes.

P2P replication allows writes to any node.

Dynamo features both sharding and P2P writes/reads.
Results

Average and 99.9 percentiles of latencies for read and write requests during peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours.

Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages.
NoSQL Use Cases

Shopping carts

User profiles and preferences

Session information
Dynamo Summary

“Eventually” consistent data store
Always writable

Decentralized

All nodes have the same responsibilities

Amazon.com’s Resolution
Weakening consistency property in the system
Increase the availability
Datastores

No-SQL
Dynamo, Apache Cassandra (column oriented as well)
BigTable (Google) (sparse, distributed multi-dimensional sorted map)
Hbase (Facebook)
Riak (Dynamo clone in Erlang)
Voldemort (LinkedIn)

P2P datastores
BitTorrent
Chord/DHT based
Freenet
PAST
GNUnet (not covered)
What is SDN?

IP networks do not have the separation of the control and data planes that is typically found in telecom solutions. There is a need to virtualize also networks (in addition to computing power and storage). Software-Defined Networking (SDN) is a new networking architecture which abstracts the logical part of computer networks to a controller. Pioneering work at Stanford and UC Berkeley, the OpenFlow protocol enables programmable control of the network and network virtualization.
Layers

In the traditional network, the control plane and the data plane are coupled together within a switch or router, and the forwarding logic is defined through internal software and protocols. In contrast, OpenFlow (or an SDN) separates this forwarding logic and moves it to an external controller. Figure 2 shows an OpenFlow-enabled switch and the controller via the OpenFlow Protocol.

1. Flow Table: An OpenFlow switch is required to have at least one flow table to perform packet lookup and forwarding. Each flow table contains a set of flow entries with associated counters and actions indicating how to process defined flows. The flow table entry is like this:

<table>
<thead>
<tr>
<th>Header Fields</th>
<th>Counters</th>
<th>Actions</th>
</tr>
</thead>
</table>

2. A Secure Channel: The channel is used to connect the remote controller and the switch. The controlling packets can be sent between the channel with OpenFlow Protocol.

3. The OpenFlow Protocol: The protocol is implemented between switches and the controller to provide a standard and programmable way for the control plane and data plane to communicate.

Each packet is processed by the switch according to the flow table:

OpenFlow Structure

**Flow Table:** An OpenFlow switch is required to have at least one flow table to perform packet lookup and forwarding. Each flow table contains a set of flow entries with associated counters and actions indicating how to process defined flows.

**A Secure Channel:** The channel is used to connect the remote controller and the switch.

**The OpenFlow Protocol:** The protocol is implemented between switches and the controller to provide a standard and programmable way for the control plane and data plane to communicate.
SDN Results

Generalizing network devices and functions. 

The vision of a network operating system. 

A network operating system is software that abstracts the installation of state in network switches from the logic and applications that control the behavior of the network.

Layering: data plane, state management plane, control logic

Distributed state management techniques

Onix: network information base shared by controller replicas
Consistency and durability
A general framework for distributed and fault-tolerant OpenFlow controller implementations

Used in massive datacenters including Google’s internal backbone

Network Information Base (NIB) abstraction (network graph)
Entities in the network topology
Entity is a key-value pair
Partitioned over Onix instances

To fill this void, in this paper we describe the design and implementation of a production-quality control platform called Onix (Sections 2-5). While we do not yet have extensive experience with Onix, the datacenter projects being built on Onix are growing rapidly, any scaling limitations of the control logic must be overcome.

The control platform should simplify the task of building management applications. This includes network connectivity, physical infrastructure, and management protocols. Onix does this in a distributive manner, by exposing a high-level and generic control logic, while allowing great generality of function, and implementation of such a control platform called Onix, and they have very distinct roles (see Figure 2.1).

The control platform must handle equipartitioning, distribute and manage functions (and other) failures gracefully. The most important challenges in building a control platform satisfying all of these requirements.

**Figure 1:**

There are four components in an Onix controlled network: managed physical infrastructure, connectivity infrastructure, Onix, and the control logic implemented by the management application. This figure depicts two Onix instances: Server 1 and Server N.

**Figure 2:**

There might be settings where optimizing control plane performance is more important than reliability, and optimizing reliability is more important than control plane performance. When faced with a tradeoff between generality and control plane performance, the control platform is the crucial enabler of the SDN paradigm. The most important challenges in building a control platform are:

- **Generality:** The control platform's API must allow functionality in a variety of contexts.
- **Scalability:** The control platform should be able to scale with the underlying network size.
- **Simplicity:** The control platform should simplify the management of the network.
- **Reliability:** The control platform must handle equipartitioning, distribute and manage failures gracefully.
- **Control Plane Performance:** The control platform should optimize the control plane performance.

There are two APIs that Onix exposes to application designers.

- **Configuration API:** This API allows application designers to configure the control logic. The configuration API is used to specify the desired behavior of the control logic.
- **Application API:** This API allows application designers to interact with the control logic. The application API is used to perform actions on the network, such as adding or removing nodes, or updating the routing table.

In such settings one might not use a general-purpose control platform, because one can only rely on a fine-tuned routing protocol. Improved reliability, one can only rely on a fine-tuned routing protocol. In such settings one might not use a general-purpose control platform, because one can only rely on a fine-tuned routing protocol.

While a number of systems following the basic paradigm of SDN have been proposed, to date there has been little published work on how to build a network control platform satisfying all of these requirements. This paper can be considered the first in-depth discussion of a NOX-like control platform.

The control platform should simplify the management of the network, not the implementation of the control logic. There might be settings where optimizing control plane performance is more important than reliability, and optimizing reliability is more important than control plane performance. When faced with a tradeoff between generality and control plane performance, the control platform is the crucial enabler of the SDN paradigm. The most important challenges in building a control platform are:

- **Generality:** The control platform's API must allow functionality in a variety of contexts.
- **Scalability:** The control platform should be able to scale with the underlying network size.
- **Simplicity:** The control platform should simplify the management of the network.
- **Reliability:** The control platform must handle equipartitioning, distribute and manage failures gracefully.
- **Control Plane Performance:** The control platform should optimize the control plane performance.

There are two APIs that Onix exposes to application designers.

- **Configuration API:** This API allows application designers to configure the control logic. The configuration API is used to specify the desired behavior of the control logic.
- **Application API:** This API allows application designers to interact with the control logic. The application API is used to perform actions on the network, such as adding or removing nodes, or updating the routing table.

There are two APIs that Onix exposes to application designers.

- **Configuration API:** This API allows application designers to configure the control logic. The configuration API is used to specify the desired behavior of the control logic.
- **Application API:** This API allows application designers to interact with the control logic. The application API is used to perform actions on the network, such as adding or removing nodes, or updating the routing table.

There are two APIs that Onix exposes to application designers.

- **Configuration API:** This API allows application designers to configure the control logic. The configuration API is used to specify the desired behavior of the control logic.
- **Application API:** This API allows application designers to interact with the control logic. The application API is used to perform actions on the network, such as adding or removing nodes, or updating the routing table.
We assume this coordination is mostly static and requires between memory and CPU usage. Control logics may prefer different trade-offs with the state (or related state) remaining untouched by other Onix instances or network elements. For such coordination, state changes that affect entities are handled by the NIB. We describe this in more detail in Section 3.

The NIB provides multiple methods for the control logic to gain access to network entities. It maintains an index of all typed entities, allowing for direct querying of a specific entity. It also supports registration for notifications on state changes or the addition/deletion of an entity. Applications can use the API to register for notifications on state changes and manipulate state, provided they have exclusive access to the NIB data structure.

The NIB provides asynchronous operations, meaning that notifications of entity arrivals and maintaining their own indices are the responsibility of the control logic. The control logic must use mechanisms external to the NIB to ensure reliability properties of the system. For example, as the control logic implements distributed coordination, race-conditions in state updates will either not exist or will be transient in nature. The control logic may then inspect the contents of the NIB and verify that the state is as expected before proceeding.

Applications can use the NIB to register for notifications on state changes or the addition/deletion of an entity. The API provides a synchronization primitive: if called for an entity, the control logic will receive a callback once the state has been pushed. After receiving the callback, the control logic may then inspect the contents of the NIB, and if the state is not as expected, it should request that the state be updated. The control logic can also use the NIB to cache state snapshots for on-demand access.

Applications can register for notifications on state changes and manipulate state. The NIB provides methods for applications to register for notifications on state changes or the addition/deletion of an entity. The API provides methods for applications to register for notifications on state changes or the addition/deletion of an entity. The control logic may then inspect the contents of the NIB, and if the state is not as expected, it should request that the state be updated. The control logic can also use the NIB to cache state snapshots for on-demand access.

We note that if the control logic implements distributed coordination, race-conditions in state updates will either not exist or will be transient in nature. The control logic may then inspect the contents of the NIB and verify that the state is as expected before proceeding.
ONIX II/III

General NIB API
query, create&destroy, access attributes, notifications, synchronize, configuration, pull

Distributing the NIB
Transactional database (consistency for slow data)
One-hop DHT for volatile network state (after Dynamo)
Link utilization etc

Uses Apache Zookeeper for distributed coordination (leader election and distributed locking)
ONIX III

Application use ONIX API to change the NIB entities

These are then reflected in controller instances that initiate OpenFlow requests

Applications need to have conflict resolution logic (as in Dynamo)
Detect and resolve conflicts, write back the proper value

Applications need to register inconsistency tolerance
Custom logic (Zookeeper) for added consistency

AS level peering possible, but original article leaves this open
Apache Zookeeper

Provides primitives for reliable distributed coordination

Suitable for storing small amounts of data that has consistent state

Master election
Group membership management
File system style API

Implementation based on atomic broadcast protocol based on Paxos distributed consensus algorithm two-phase commit protocol:
  First propose a value and receive ack from majority
  After majority ack the leader broadcasts commit
SDN for Mobile Networks

NOX design appears relevant also for mobile networks
  WiFi networks
  Cellular networks

Control plane over DHT?

Overview of recent developments:
Overlays and Networking

Overlays can be used to overcome network problems
Reliable Overlay Network (RON) from 2001

Many SDN controllers are overlays with distributed state
DHTs can be used

Overlays + SDN allow more flexibility and application control
over the routing and forwarding

Testbeds are now being created