Key Value Storage

- Interface
  - `put(key, value);` // insert/write “value” associated with “key”
  - `value = get(key);` // get/read data associated with “key”

- Abstraction used to implement
  - File systems: value content → block
  - Sometimes as a simpler but more scalable “database”

- Can handle large volumes of data, e.g., PBs
  - Need to distribute data over hundreds, even thousands of machines

Key Values: Examples

- Amazon:
  - Key: customerID
  - Value: customer profile (e.g., buying history, credit card, ..)

- Facebook, Twitter:
  - Key: UserID
  - Value: user profile (e.g., posting history, photos, friends, …)

- iCloud/iTunes:
  - Key: Movie/song name
  - Value: Movie, Song

- Distributed file systems
  - Key: Block ID
  - Value: Block

System Examples


- Amazon
  - Dynamo: internal key value store used to power Amazon.com (shopping cart)
  - Simple Storage System (S3)

- BigTable/HBase/Hypertable: distributed, scalable data storage

- Cassandra: “distributed data management system” (Facebook)

- Memcached: in-memory key-value store for small chunks of arbitrary data (strings, objects)

- eDonkey/eMule: peer-to-peer sharing system
**Key Value Store**

- Also called a Distributed Hash Table (DHT)
- Main idea: partition set of key-values across many machines

**Challenges**

- **Fault Tolerance**: handle machine failures without losing data and without degradation in performance
- **Scalability**:
  - Need to scale to thousands of machines
  - Need to allow easy addition of new machines
- **Consistency**: maintain data consistency in face of node failures and message losses
- **Heterogeneity** (if deployed as peer-to-peer systems):
  - Latency: 1ms to 1000ms
  - Bandwidth: 32Kb/s to 100Mb/s

**Key Questions**

- `put(key, value)`: where do you store a new (key, value) tuple?
- `get(key)`: where is the value associated with a given "key" stored?

- And, do the above while providing
  - Fault Tolerance
  - Scalability
  - Consistency

**Directory-Based Architecture**

- Have a node maintain the mapping between **keys** and the **machines (nodes)** that store the **values** associated with the **keys**
Directory-Based Architecture

• Have a node maintain the mapping between keys and the machines (nodes) that store the values associated with the keys.

Directory-Based Architecture

• Having the master relay the requests → recursive query
• Another method: iterative query (this slide)
  – Return node to requester and let requester contact node

Discussion: Iterative vs. Recursive Query

• Recursive Query:
  – Advantages:
    » Faster, as typically master/directory closer to nodes
    » Easier to maintain consistency, as master/directory can serialize puts()/gets()
  – Disadvantages: scalability bottleneck, as all “Values” go through master/directory

• Iterative Query
  – Advantages: more scalable
  – Disadvantages: slower, harder to enforce data consistency
Fault Tolerance

- Replicate value on several nodes
- Usually, place replicas on different racks in a datacenter to guard against rack failures

Master/Directory

\[ \text{Master/Directory} \]

\[ \text{put}(K_{14}, V_{14}) \]

\[ N_1, N_3 \]

Fault Tolerance

- Again, we can have
  - Recursive replication (previous slide)
  - Iterative replication (this slide)

Master/Directory

\[ \text{Master/Directory} \]

\[ \text{put}(K_{14}, V_{14}) \]

\[ N_1, N_3 \]

Scalability

- Storage: use more nodes

Request Throughput:
  - Can serve requests from all nodes on which a value is stored in parallel
  - Master can replicate a popular value on more nodes

Master/directory scalability:
  - Replicate it
  - Partition it, so different keys are served by different masters/directories
    » How do you partition? (p2p DHDT, end of semester)

Scalability: Load Balancing

- Directory keeps track of the storage availability at each node
  - Preferentially insert new values on nodes with more storage available

- What happens when a new node is added?
  - Cannot insert only new values on new node. Why?
  - Move values from the heavy loaded nodes to the new node

- What happens when a node fails?
  - Need to replicate values from fail node to other nodes
Replication Challenges

• Need to make sure that a value is replicated correctly
• How do you know a value has been replicated on every node?
  – Wait for acknowledgements from every node
• What happens if a node fails during replication?
  – Pick another node and try again
• What happens if a node is slow?
  – Slow down the entire put()? Pick another node?
• In general, with multiple replicas
  – Slow puts and fast gets

Consistency

• How close does a distributed system emulate a single machine in terms of read and write semantics?

  • Q: Assume put(K14, V14') and put(K14, V14'') are concurrent, what value ends up being stored?
    • A: assuming put() is atomic, then either V14' or V14'', right?
  
  • Q: Assume a client calls put(K14, V14) and then get(K14), what is the result returned by get()?
    • A: It should be V14, right?

  Above semantics, not trivial to achieve in distributed systems

Concurrent Writes (Updates)

• If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order

  • put(K14, V14') and put(K14, V14'') reach N1 and N3 in reverse order
  • What does get(K14) return?
    • Undefined!
Read after Write

- Read not guaranteed to return value of latest write
  - Can happen if Master processes requests in different threads

```
put(K14, V14')
get(K14) happens right after put(K14, V14')
get(K14) reaches N3 before put(K14, V14')!
```

Consistency (cont’d)

- Large variety of consistency models:
  - Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image)
    - Think "one updated at a time"
    - Transactions (later in the class)
  - Eventual consistency: given enough time all updates will propagate through the system
    - One of the weakest form of consistency; used by many systems in practice
  - And many others: causal consistency, sequential consistency, strong consistency, …

Strong Consistency

- Assume Master serializes all operations
- Challenge: master becomes a bottleneck
  - Not address here
- Still want to improve performance of reads/writes \(\rightarrow\) quorum consensus

Quorum Consensus

- Improve put() and get() operation performance
- Define a replica set of size N
- put() waits for acks from at least W replicas
- get() waits for responses from at least R replicas
- \(W+R > N\)
- Why does it work?
  - There is at least one node that contains the update
- Why you may use \(W+R > N+1\)?
Quorum Consensus Example

• N=3, W=2, R=2
• Replica set for K14: (N1, N2, N4)
• Assume put() on N3 fails

Conclusions: Key Value Store

• Very large scale storage systems

Quiz 15.1: Key-Value Store

• Q1: True _ False _ On a single node, a key-value store can be implemented by a hash-table
• Q2: True _ False _ Master can be a bottleneck point for a key-value store
• Q3: True _ False _ Iterative puts achieve lower throughput than recursive puts
• Q4: True _ False _ With quorum consensus, we can improve read performance at expense of write performance
Quiz 15.1: Key-Value Store

• Q1: True _ False _ On a single node, a key-value store can be implemented by a hash-table
• Q2: True _ False _ Master can be a bottleneck point for a key-value store
• Q3: True _ False _ Iterative puts achieve lower throughput than recursive puts
• Q4: True _ False _ With quorum consensus, we can improve read performance at expense of write performance

Networking: This Lecture’s Goals

• What is a protocol?
• Layering

What Is A Protocol?

• A protocol is an agreement on how to communicate

• Includes
  – Syntax: how a communication is specified & structured
    » Format, order messages are sent and received
  – Semantics: what a communication means
    » Actions taken when transmitting, receiving, or when a timer expires
Examples of Protocols in Human Interactions

- Telephone
  1. (Pick up / open up the phone.)
  2. Listen for a dial tone / see that you have service.
  3. Dial
  4. Should hear ringing …
  5. Callee: “Hello?”
  6. Caller: “Hi, it’s Alice ….”
     Or: “Hi, it’s me” (← what’s that about?)
  7. Callee: “Hey, do you think … blah blah blah …” pause
  8. Callee: “Yeah, blah blah blah ….” pause
  9. Caller: Bye
 10. Callee: Bye
 11. Hang up

Examples of Protocols in Human Interactions

Asking a question

1. Raise your hand.
2. Wait to be called on.
3. Or: wait for speaker to pause and vocalize

End System: Computer on the ‘Net

Also known as a “host”…

Clients and Servers

- Client program
  - Running on end host
  - Requests service
  - E.g., Web browser

GET /index.html
Clients and Servers

- **Client program**
  - Running on end host
  - Requests service
  - E.g., Web browser

- **Server program**
  - Running on end host
  - Provides service
  - E.g., Web server

```
GET /index.html
```

Site under construction

---

Client-Server Communication

- **Client**
  - Sometimes on
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn’t communicate directly with other clients
  - Needs to know the server’s address

- **Server**
  - Always on
  - Services requests from many client hosts
  - E.g., Web server for the www.cnn.com Web site
  - Doesn’t initiate contact with the clients
  - Needs a fixed, well-known address

---

Peer-to-Peer Communication

- Not always-on server at the center of it all
  - Hosts can come and go, and change addresses
  - Hosts may have a different address each time

- Example: peer-to-peer file sharing (e.g., Bittorrent)
  - Any host can request files, send files, query to find where a file is located, respond to queries, and forward queries
  - Scalability by harnessing millions of peers
  - Each peer acting as both a client and server

---

The Problem

- Many different applications
  - Email, web, P2P, etc.

- Many different network styles and technologies
  - Wireless vs. wired vs. optical, etc.

- How do we organize this mess?
The Problem (cont’d)

• Re-implement every application for every technology?
• No! But how does the Internet design avoid this?

Solution: Intermediate Layers

• Introduce intermediate layers that provide set of abstractions for various network functionality & technologies
  – A new app/media implemented only once
  – Variation on “add another level of indirection”

Software System Modularity

Partition system into modules & abstractions:
• Well-defined interfaces give flexibility
  – Hides implementation - thus, it can be freely changed
  – Extend functionality of system by adding new modules
• E.g., libraries encapsulating set of functionality
• E.g., programming language + compiler abstracts away not only how the particular CPU works …
  – … but also the basic computational model
• Well-defined interfaces hide information
  – Isolate assumptions
  – Present high-level abstractions
  – But can impair performance

Network System Modularity

Like software modularity, but:
• Implementation distributed across many machines (routers and hosts)
• Must decide:
  – How to break system into modules
    » Layering
  – What functionality does each module implement
    » End-to-End Principle
• We will address these choices next lecture
Layering: A Modular Approach

- Partition the system
  - Each layer solely relies on services from layer below
  - Each layer solely exports services to layer above

- Interface between layers defines interaction
  - Hides implementation details
  - Layers can change without disturbing other layers

Protocol Standardization

- Ensure communicating hosts speak the same protocol
  - Standardization to enable multiple implementations
  - Or, the same folks have to write all the software

- Standardization: Internet Engineering Task Force
  - Based on working groups that focus on specific issues
  - Produces "Request For Comments" (RFCs)
    - Promoted to standards via rough consensus and running code
  - IETF Web site is http://www.ietf.org
  - RFCs archived at http://www.rfc-editor.org

- De facto standards: same folks writing the code
  - P2P file sharing, Skype, <your protocol here>…

Example: The Internet Protocol (IP): “Best-Effort” Packet Delivery

- Datagram packet switching
  - Send data in packets
  - Header with source & destination address

- Service it provides:
  - Packets may be lost
  - Packets may be corrupted
  - Packets may be delivered out of order

Example: Transmission Control Protocol (TCP)

- Communication service
  - Ordered, reliable byte stream
  - Simultaneous transmission in both directions

- Key mechanisms at end hosts
  - Retransmit lost and corrupted packets
  - Discard duplicate packets and put packets in order
  - Flow control to avoid overloading the receiver buffer
  - Congestion control to adapt sending rate to network load
Quiz 15.2: Protocols

• Q1: True _ False _ Protocols specify the syntax and semantics of communication
• Q2: True _ False _ Protocols specify the implementation
• Q3: True _ False _ Layering helps to improve application performance
• Q4: True _ False _ "Best Effort" packet delivery ensures that packets are delivered in order
• Q5: True _ False _ In p2p systems a node is both a client and a server
• Q6: True _ False _ TCP ensures that each packet is delivered within a predefined amount of time

Quiz 15.2: Protocols

• Q1: True _ False _ Protocols specify the syntax and semantics of communication
• Q2: True _ False _ Protocols specify the implementation
• Q3: True _ False _ Layering helps to improve application performance
• Q4: True _ False _ "Best Effort" packet delivery ensures that packets are delivered in order
• Q5: True _ False _ In p2p systems a node is both a client and a server
• Q6: True _ False _ TCP ensures that each packet is delivered within a predefined amount of time

Summary

• Roles of
  – Standardization
  – Clients, servers, peer-to-peer
• Layered architecture as a powerful means for organizing complex networks
  – Though layering has its drawbacks too
• Next lecture
  – Layering
  – End-to-end arguments