# Distributed Systems

08r. Assignment 4 Review

David Domingo
Paul Krzyzanowski
Rutgers University
Fall 2018



## Question 1

What are three advantages of using a large chunk size in GFS?

1. Reduce interactions with master
   It reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information. Large chunk ⇒ fewer queries
2. Reduce network overhead
   Since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time.
3. Reduces the size of the metadata stored on the master
   A master has fewer chunks to keep track of per file.



## Question 2

Explain the role of the GFS master.

The master maintains all file system metadata.

- This includes
  - the namespace
  - access control information
  - mapping from files to chunks
  - current locations of chunks
- It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers.



## Question 3

As Dropbox's design evolved, why did Dropbox split the original web server into two web servers? [What was the function of each server?]

- Dropbox ran out of capacity at the server because all uploads and downloads went to one server.
  - One server dealt with metadata.
  - Another dealt with file uploads and downloads.



## Question 4

Why were notification servers added?

Notification servers were added to not require clients to poll the server to check for changes.

This reduces the load on the system since clients that don't have changes don't impose any traffic onto the servers.



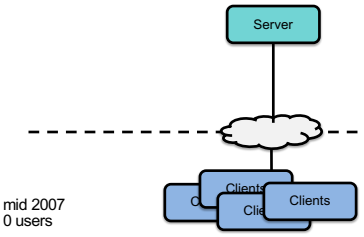
## Question 5

Why was RPC-based communication added to the blockservers instead of having them talk to the database?

- Avoids multiple round-trip calls from the blockserver to the database: RPCs can contain higher-level commands that can be processed at the blockservers directly.

## Dropbox: architecture evolution: version 1

One server: web server, app server, mySQL database, sync server



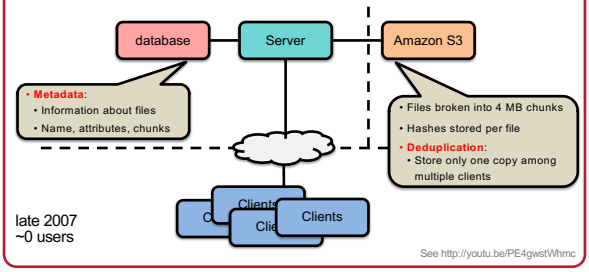

mid 2007
0 users

See http://youtu.be/PE4gwstWhmc



## Dropbox: architecture evolution: version 2

- Server ran out of disk space:
  moved data to Amazon S3 service (key-value store)
- Servers became overloaded: moved mySQL DB to another machine
- Clients periodically polled server for changes



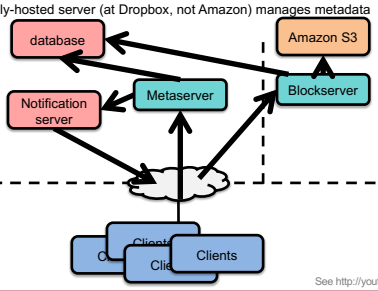

late 2007
~0 users

See http://youtu.be/PE4gwstWhmc



## Dropbox: architecture evolution: version 3

- Move from polling to notifications: add **notification server**
- Split web server into two:
  - Amazon-hosted server hosts file content and accepts uploads (stored as blocks)
  - Locally-hosted server (at Dropbox, not Amazon) manages metadata



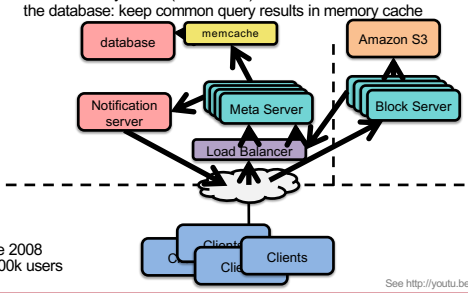

early 2008
50k users

See http://youtu.be/PE4gwstWhmc



## Dropbox: architecture evolution: version 4

- Add more *metaservers* and *blockservers*
- Blockservers do not access DB directly; they send RPCs to metaservers
- Add a memory cache (memcache) in front of the database to avoid scaling the database: keep common query results in memory cache



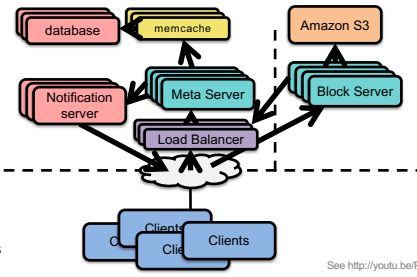

late 2008
~100k users

See http://youtu.be/PE4gwstWhmc



## Dropbox: architecture evolution: version 5

- 10s of millions of clients: Clients have to connect before getting notifications
- Add 2-level hierarchy to notification servers: ~1 million connections/server




early 2012
>50M users

See http://youtu.be/PE4gwstWhmc



## Exam 2 Selected Questions & Answers

## Fall 2017 Question 1

Explain what happens in each of the two phases of a *two phase commit protocol* (just the main points; no details about what to log).

**Phase 1: solicit votes**

Coordinator sends a message to all participants asking if they can commit. Wait for ***all*** responses (as long as necessary)

**Phase 2: Send commit or abort directive**

If there is ***unanimous*** agreement to commit, then the coordinator sends a *commit* directive to ***all*** participants; otherwise it sends an *abort* directive to ***all*** participants

Wait for ***all*** participants to acknowledge



## Fall 2016: Question 2

Why is a transaction log crucial in providing fault tolerance in a two-phase commit protocol?
*Hint: the answer has nothing to do with aborts*

It allows the transaction to continue from where it left off when the system restarts.

A transaction cannot change its mind even if it dies and restarts. The log keeps the transaction's state.

*Bad answers:*

- *"Revert if a sub-transaction fails" or "enable rollback"*
  - *That's only done in the case of aborts*
- *Recovery server can take over*
  - *A recovery server cannot access a failed coordinator's log*



## Fall 2017 Question 2

Explain why Eric Brewer's CAP theorem led to the use of an *eventual consistency* model in many distributed systems.

Given that ***partitions*** are a fact of life in production systems, the CAP theorem states that we have to choose ***availability*** or ***consistency***.

Many services value availability over consistency.

Eventual consistency means that some copies of data will be stale but updates will eventually propagate to all replicas.



## Fall 2017 Question 3

*False deadlock* cannot be solved simply by imposing total ordering on messages. Explain why.

Total ordering ensures consistent ordering at all receivers. *Here, we have only one receiver.*

Total ordering does not guarantee global time ordering. Messages may still arrive out of order

A *lock request* message may reach a sequence # server before a *lock release* message even if the *release* was sent first.

-1 for process might die: that's a general problem beyond detecting false deadlock (and you have to consider the recovery model: if it's fail-recover, you may not want to steal the lock).



## Fall 2016: Question 4

How did callbacks in AFS enable it to scale to support more clients than NFS?

They allow each client to support long-term caching – no need to check with the server; it will tell you if a file has been modified.

*Bad answer:*

*Explain what callbacks are without explaining why they enable AFS to scale*



## Fall 2016: Question 5

Under what conditions would consistent hashing be unnecessary for a distributed hash table?

If you never need to expand or shrink the table – i.e., you never need to add or remove servers

The whole purpose of consistent hashing is to create hash keys that don't change if you change the size of the table.

*Bad answers:*

- *"If there are fewer keys than the number of slots"*
- *"If there are no collisions" or any answer related to collisions.*

## Fall 2016 – Exam 3 – Question 1

Unlike the design of Chord, Amazon Dynamo stores the entire list of nodes on each system. Explain the advantage of doing this instead of using a finger table.

Each node knows exactly which node contains the desired key and can forward the query there directly. There is no need for multiple hops.

A finger table may require hops: O(log N) vs. O(1)

*Bad answers:*

- *Better fault tolerance*
- *Easier to add new nodes*
  *(if every node stores the entire list, you still need to update all nodes)*



## Fall 2017 Question 4

As opposed to locks, *leases*:
(a) Have expiration times.
(b) Are long-term while locks are short-term.
(c) Allow multiple clients to access a resource for reading.
(d) Can be distributed while locks are centralized.

Leases are just locks that expire.



## Fall 2017 Question 5

*Ricart & Agrawala's* mutual exclusion algorithm differs from Lamport's because it:
(a) Uses more messages than Lamport's since each message needs to be acknowledged.
(b) Uses fewer messages than Lamport's since a system does not reply until it agrees to grant access to a resource.
(c) Does not require the use of Lamport timestamps.
(d) Requires a system to contact all group members to request access to a resource.

(a) Lamport's actually may use more messages since all messages are acknowledged all messages immediately & *release* messages are sent to all members. Instead of sending a *release* message, Ricart & Agrawala's delays on responding with an *ack*.

(c) Both use unique Lamport timestamps.

(d) Both require sending messages to the entire group.



## Fall 2017 Question 6

A *bully election algorithm* elects:
(a) The client who can send the most messages in a given time interval.
(b) The client who discovered a dead leader and sends the first election message.
(c) The client who has the highest-numbered process ID and responds to messages.
(d) The client that gets a majority vote.

Contact all higher-numbered processes.

If any respond, you're done: someone above you is the winner – let them figure it out.

If not, then you're the winner.



## Fall 2017 Question 7

One advantage that the *Chang and Roberts algorithm* has over the ring algorithm is:
(a) It is guaranteed to complete.
(b) It is resistant to partitioning.
(c) It will never allow concurrent elections to pick different winners.
(d) It creates smaller messages.

Only one process ID is sent with an election message – not a list.



## Fall 2017 Question 8

If an acceptor starts up in the middle of the Paxos protocol, how does it get information about the current proposal?
(a) A proposer will send it during the second (accept) phase.
(b) It queries all other live acceptors for the number.
(c) It queries any other live acceptor for the number.
(d) A learner will send it the last valid proposal number.

Proposers choose values (they're active) – contact a majority of acceptors

Phase 1 (PREPARE-PROMISE): if a majority of acceptors respond with a promise, then a proposer knows it can proceed. The proposer gets either its proposla # or, if another proposal was already accepted, the proposal # & value of the accepted proposal

Phase 2: (PROPOSE-ACCEPT): Of all the responses from the acceptors, the proposer again contacts the **majority** of acceptors and asks each one to accept the value with the highest numbered proposal (or its own if there were no others)

Acceptors are passive elements – they only respond to messages from proposers and store the chosen value – accepting a new value if it has a higher (newer) proposal #

Learners just propagate results at the end.

## Fall 2017 Question 9

Once an acceptor makes a *promise* on a received proposal, it will:
(a) Not accept proposals with higher sequence numbers.
(b) Not accept proposals with lower sequence numbers.
(c) Not accept any other incoming proposals.
(d) Accept any future proposals, regardless of their number.

Not accepting lower sequence numbers means that delayed messages will never cause a majority of acceptors to revert to an older proposal.

An acceptor may accept higher sequence numbers – change its mind.

A proposer must get PROMISE messages from a **majority** of acceptors.

It then must send a PROPOSE message with the value of the highest proposal to a **majority** of acceptors



## Fall 2017 Question 10

One way in which *Raft* differs from Paxos is that:
(a) There is no need for majorities.
(b) In some cases, the resultant value might not be one of the proposed values.
(c) A single leader to receive all client requests is a requirement.
(d) The protocol might fail to achieve consensus and will need to be restarted.

(a) Raft relies on overlapping majorities to guarantee safety: allows the Raft cluster to continue operating during membership changes. So does Paxos.
- *A majority of members must elect a leader.*

- Log entries (build into the algorithm) must reach a majority of members

(b, d) Then it wouldn't be a valid consensus algorithm.

(c) A single elected leader handles all client requests. This is *optional* in Paxos; Paxos can run with an arbitrary number of proposers.



## Fall 2017 Question 11

When Raft servers hold an *election*, the winner is generally:
(a) The server with the highest process number.
(b) The server that picks the highest random number.
(c) Chosen by a leader, who propagates the choice to a majority of followers.
(d) The server where a majority of the group members receive its election message first.

To start an election:

- A candidate picks a random election timeout
- It then votes for itself and requests votes from the entire group
- If a candidate receives a *request vote* message and hasn't yet voted for itself, it picks the candidate that sent the message & responds
- When a candidate gets a majority of votes, it becomes the leader



## Fall 2017 Question 12

The *three-phase commit protocol* inserts a new phase to:
(a) Give a participant the chance to change its mind about committing.
(b) Tell a participant the vote but not have it commit its sub-transaction.
(c) Tell each participant that it can release any locks it has on resources.
(d) Ask a participant if it is ready to commit or needs to abort.

3PC is designed to make it easy to have a replacement coordinator take over.

By propagating the vote to all participants, the coordinator can ask *any* participant for the state of the vote:

- If the participant doesn't know the vote, then nobody has been told to commit or abort and we can restart the protocol.
- If the participant knows the vote, then we know there was unanimous agreement
- If a participant already committed or aborted, we know there was unanimous agreement



## Fall 2017 Question 13

A problem with two-phase locking that is fixed by *strict two-phase locking* is:
(a) Since locks are advisory, other transactions may be able to access that locked data.
(b) A transaction could read data that was modified by a transaction that did not yet commit.
(c) Deadlock can occur.
(d) The lock manager may die between the first and second phase.

(a) Locks are not advisory; they are mandatory.

(b) Transaction #2 can read data that transaction #1 has unlocked before transaction #1 commits. If transaction #1 aborts, transaction #2 will have to abort ⇒ cascading aborts

(c) Just as possible with strict 2PL

(d) Just as possible with strict 2PL – use a fault-tolerant lock manager



## Fall 2017 Question 14

The use of separate *read locks* and *write locks*:
(a) Allows multiple transactions to acquire write locks to write the same resource concurrently.
(b) Allows multiple transactions to acquire read locks to read the same resource concurrently.
(c) Is a form of two-phase locking that separates locks based on their type.
(d) All of the above.

No harm done with concurrent reads if nobody is modifying.

Read locks keep out writers.

(a) No – only one writer at a time.

(c) Read & write locks are not a form of two-phase locking.

## Fall 2017 Question 15

What condition is *NOT* necessary for *deadlock*?
(a) A transaction holding exclusive locks on one or more resources.
(b) The ability for a transaction to preempt another one to obtain a lock.
(c) A transaction waiting on locks for one or more resources.
(d) A circular dependency of transactions waiting for locks on resources.

Conditions for deadlock:

1. Mutual exclusion
2. Non-preemption
3. Hold & wait
4. Circular dependency



## Fall 2017 Question 16

*Edge chasing* is a technique to:
(a) Make sure that release messages are delivered before lock messages.
(b) Allow older transactions to complete before new ones start.
(c) Schedule transactions so that they will never access the same resource concurrently.
(d) Determine if a cycle of waiting on resource locks exists.

*Edge chasing = sending probe messages to processes holding resources you want … and seeing if the messages come back to you.*



## Fall 2017 Question 17

NFS's *validation* technique ensures that cached data is purged when:
(a) The server informs the client that its cached contents are no longer valid.
(b) The client releases a lock on the remote file.
(c) The client closes the file.
(d) The server is contacted for new data and the client discovers the file's modification time changed.

Each time a remote request is made, the client checks the modification time of the remote file.



## Fall 2017 Question 18

*Session semantics* on a file mean that:
(a) Only one client can read a file at a time.
(b) You can grab a single lock to get exclusive access to multiple files at the same time.
(c) Writes from multiple clients will occur in the order in which they were issued.
(d) Nobody sees your writes until you close the file.

Your modifications are not visible to others until you close the file. The last process to close a file overwrites all other changes.



## Fall 2017 Question 19

Under the Coda file system, a client has to write changes to:
(a) A master server, which then propagates them to replicas.
(b) Any server hosting the volume, which will then propagate them to other replicas.
(c) The client modification log, which are then sent to the cell directory server.
(d) All available servers with a copy of the volume.

Clients are responsible for updating all replicas.



## Fall 2017 Question 20

A Coda *Client Modification Log* (*CML*) is used by:
(a) Servers to inform clients which files have been modified by other clients.
(b) Servers to keep a log of which clients have made changes to files.
(c) Clients to upload a list of changes to a file instead of uploading the entire file to a server.
(d) Clients during periods of disconnection to track which files have been modified.

(a, b): CML is used by clients

(c): It doesn't store changes to files

(d): Just a list of modification files

## Fall 2017 Question 21

SMB *oplocks*:
(a) Allow the client to lock a region of a remote file.
(b) Provide a general-purpose distributed mutual exclusion service for files and other resources.
(c) Are a mechanism for the server to lock clients out while changes are being made to a file.
(d) Tell the client how it can cache file data.

(a) They're not locks but caching directive

(b) Nope.

(c) Nope.



## Fall 2017 Question 22

A key to *Chubby's* good performance is:
(a) Distributing data across multiple replicas.
(b) Using exceptionally large block sizes in its file system.
(c) Caching all file data in memory on the server.
(d) Disallowing clients from locking any file managed by Chubby.

(a) Replication is only for fault tolerance. One server handles all requests at a time.

(b) Nope. That's GFS.

(c) Everything lives in memory - and stored for persistance.

(d) Nothing to do with performance – but Chubby supports advisory file locking.



## Fall 2017 Question 23

A *notification server* in Dropbox is conceptually similar to:
(a) SMB oplocks.
(b) AFS callbacks.
(c) NFS read-aheads.
(d) GFS master.

A notification server avoids the need for clients to check if any locally-stored files have been modified.



## Fall 2017 Question 24

A design aspect of *parallel file systems* is:
(a) File data is spread across multiple servers.
(b) Multiple clients can access the same file data concurrently.
(c) The file system uses huge block sizes.
(d) The same file data is replicated on multiple servers.



## Fall 2017 Question 25

GFS separates *data flow* from *control flow* to:
(a) Reduce the amount of time that a chunk needs to be locked.
(b) Allow the master to decide which chunkservers will host which replicas.
(c) Enable clients to write data at the same time that the master handles file creation.
(d) Enable processing to take place on the data as it is being written to the servers.

It takes a while to upload megabytes of data, so don't lock the file then.

Get the data to the servers first … then lock the file & update it … making sure that all replicas process concurrent updates in the same order.



## Fall 2017 Question 26

*Consistent hashing* means:
(a) The hash result will never be greater than the table size.
(b) A hash function on a key, *H(k)*, returns the same value each time.
(c) Most keys will not have to be remapped if the table size changes.
(d) The hash function can only accept valid keys.

(a, b): This is true of any hash function / hash table.

(d) This doesn't really make sense.

## Fall 2017 Question 27

In a Chord ring with 16 nodes, each node would need a *finger table* with this many entries:
(a) 4
(b) 5
(c) 15
(d) 16

Entry 0: neighboring node – $2^0$ = 1 hop away
Entry 1: $2^1$ = 2 hops away
Entry 2: $2^2$ = 4 hops away
Entry 3: $2^3$ = 8 hops away
} 4 entries

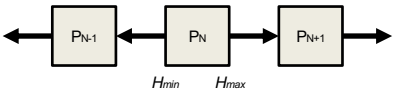
*Entry 4: $2^5$ = 16 hops away … too far!*



## Fall 2017 Question 28

The average hop count in a **one-dimensional** content-addressable network of *N* nodes is:
(a) *N/2*
(b) $\sqrt{N}$
(c) $\log_2 N$
(d) $N^2$

Each process handles a range of hash values and knows of left and right neighbors – a linear search is needed





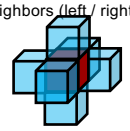

## Fall 2017 Question 29

Each node in a three-dimensional content-addressable network must know about at most this many neighbors:
(a) 2
(b) 4
(c) 6
(d) 8

1-dimensional CAN: 2 neighbors (left / right)

2-dimensional CAN: 4 neighbors (left / right / top / bottom)

3-dimensional CAN: 6 neighbors (left / right / top / bottom / front / back )





## Fall 2017 Question 30

*Virtual nodes* in Dynamo:
(a) Enable the use of virtual machines that can be brought in to handle extra load during peak traffic times.
(b) Are empty placeholder nodes in the ring between physical successor nodes.
(c) Allow control of load distribution by assigning varying numbers of virtual nodes to physical nodes.
(d) Each manage one logical block of a content-addressable network.



## Fall 2017 Question 31

Dynamo's *optimistic replication* means:
(a) Dynamo uses rack-aware logic to create replicas on systems closest to the original node.
(b) Applications can safely assume that their data will be replicated.
(c) Replicas are not guaranteed to be identical at all times.
(d) Applications can assume that all replicas will be updated atomically, ensuring ACID semantics.

Dynamo employs an eventually consistent model



The end