RPC original 1984 proposed paper

- function implementation lives on server
- client wants to execute it.
- goal: transparent. looks like executing on own machine
- client: sends msg: marshalled into binary transmitable package.
- [[42e4cce2f4aae747c5e3f34e188484cd_MD5.jpeg|Open: Pasted image 20241020135520.png]] ![[42e4cce2f4aae747c5e3f34e188484cd_MD5.jpeg]]
- [[97e3959828ef355c67e869b7f4648648_MD5.jpeg|Open: Pasted image 20241020135403.png]] ![[97e3959828ef355c67e869b7f4648648_MD5.jpeg]]
- server: unmarshals msg, passes to app, sends fn(x,y) back to client

Example Javascript RPC < skip examples 12:00-16:00

```
let args = {amount: 3.99, currency: 'GBP', /*...*/ }; //f(x,y)
let request = {
    method: 'POST',
    body: JSON.stringify(args),
    headers: {'Content-Type': 'application/json'}
};
fetch('https://example.com/payments', request)
    .then((response) => { //check status
        if (response.ok) success(response.json());
        else failure(response.status); // server error
})
.catch((error) => {
    failure(error); // network error
});
```

RPC Benefits - easy to use, familiar to programmers - abstracts away network/marshal details in raw network sockets, endianess ## Failures - client or server crash/reboot - packet network loss, routing issues - slow network/server all look same to client

Three failure-handling schemas - At-Least-Once - client gets no ACK, retry fixed # times until give up and ret error - **read-only**, otherwise retry misinterpreted by server as multiple requests - **idempotent**: or request can only happen once (i.e. create account) - At-Most-Once - server responsible for detecting and rejecting dupe responses - transaction ID (XID) - client ID (IP) + time of day //prevents parallelism - unique ID + seq no // using IP bad unless small network - random # (practically doesn’t work) - issues: - seen XIDS grow without bound!!!! - soln: on next client request, client specifies upper bound of seqno. “I saw up to seqno N” and server deletes all responses and their XIDs < N - dupe while still executing original? - server crash/restart? - soln: routinely saves response & XID buffer to disk - granularity of backups (1sec, 10sec, etc) based on laboriousness of handling request

```
    if seen[xid]:
        retval = old[xid] // get ret val in buffer of old responses
    else:
        retval = handler()
        old[xid] = retval //cache response in buffer
        seen[xid] = true //mark as seen before

    return retval
```

- Exactly-Once (ideal but hard to implement)
    - client retries if no ACK
    - server filters out seen responses
    - server backups to disk

MapReduce

- managing lots of big data records (i.e. books, websites) and easily compute some metric over all records
- records all same format (i.e. csv, xml, json, tsv)
    - i.e. html file: teach mapreduce what indicates start/end of each record
    - i.e. csv file: each record its own line
- key:value pair, key not unique.
- sort all lambs into groups by key, compute summarizing func over all groups/keys
    - key: The value: count over all words

Example:

MAP: (k1,v1) –> list(k2,v2) - two records: - mary had a little lamb little lamb little lamb - the quick mary jumps over the lazy lamb - each record: per word occurrence, key-val pair and store on RAM –> disk - {'Mary': 1, 'had': 1, 'little': 1, 'lamb': 1, 'little': 1, 'lamb': 1, 'little': 1, 'lamb': 1} - {'the': 2, 'quick': 1, 'mary': 1, 'jumps': 1, 'over': 1, 'lazy': 1, 'lamb': 1} - combine all sets into one without removing dupes - {'Mary': 1, 'had': 1, 'little': 1, 'lamb': 1, 'little': 1, 'lamb': 1, 'little': 1, 'lamb': 1, 'the': 2, 'quick': 1, 'mary': 1, 'jumps': 1, 'over': 1, 'lazy': 1, 'lamb': 1} - partition (key=word, val=list), intermediate key-value pairs - { 'Mary': [1], 'had': [1], 'little': [1, 1, 1], 'lamb': [1, 1, 1, 1], 'the': [1], "a": [1], 'mary': [1, 1], 'jumps': [1], 'over': [1], 'lazy': [1] } - sort by keys to easily send key-val pairs to proper reducer (for efficiency) - assume each key hashes to unique val - (sort so can send chunks instead of iterating) REDUCE: (k2, list(v2)) –> list(v2) - reduce/summarizing function on list of vals for each key - { 'Mary': 1, 'had': 1, 'little': 3, 'lamb': 4, 'the': 1, "a": 1, 'mary': 2, 'jumps': 1, 'over': 1, 'lazy': 1 }

vs divide-and-conquer: - splits data to chunks but NOT recursively - share-nothing architecture: each map task independent - split-apply-combine alg - embarrassingly parallel: super easy to parallelize

Use cases

- distributed grep. to find all records matching pattern
    - map: (record ID, True) or even better, omit any false.
    - reduce: trivial, identity reducer (takes list and outputs list)
- count # times url access (input=urls)
    - map: (url, 1), (url2, 1), … (urlN, 1)
    - reduce: sum up values by url
- reverse web-link graph (edge list U->V, find all pages u linking to v)
    - map: (v,u)
        - ex: (PageA, PageB) (PageC, PageB) (PageD, PageE) (PageF, PageE)
        - reverse order to get V->U
            - (PageB, PageA) (PageB, PageC) (PageE, PageD) (PageE, PageF)
        - combine same keys
            - (PageB, [PageA, PageC])
            - (PageE, [PageD, PageF])
    - reduce: trivial.
- term vector per host: key=url, val=content, word counter per url
- inverted index: key=url, val=content, list of urls containing each word
- distrib sort: given key-value pairs, sort by key SEE DISCUSSION SECTION

System Implementation (schedule, parallelize)

- each job = map+reduce scheduled by scheduler
- before processing, M splits = # map tasks paralellized in map phase
- assume program standalone (map+reduce+config)
- old implementation: map finish before can reduce, unlike Hadoop

[[7ab2b8f3912bf46740d42bc73866794c_MD5.jpeg|Open: Pasted image 20241020151917.png]] ![[7ab2b8f3912bf46740d42bc73866794c_MD5.jpeg]] [[b0b01bd7646cfe20af0860ee9b0e9c2c_MD5.jpeg|Open: Pasted image 20241020152535.png]] [[a2de891b8b2abab4cc6689f0e9875633_MD5.jpeg|Open: Pasted image 20241020153019.png]] ![[a2de891b8b2abab4cc6689f0e9875633_MD5.jpeg]] 1. user spawns master 2. master spawns workers 3. master assigns workers to be mapper or reducer or both and where/which data they in charge of (splits) 4. mapping workers take assigned split and writes intermediate key-val pairs to disk 5. while map running, reducers pull from intermediate key-val pairs in RAM/disk but CANNOT REDUCE (as still dependent) 6. when map done, worker notifies master and where output is 7. master tells same/diff worker to reduce task with intermediate output. 8. reducer makes RPC call to mapper and request sorted partition of intermediate output from mapper 9. reducer writes output to outpute fileS (one per reducer) and done 10. why one per reducer? easily pipe into another mapreducer job

master: locations of intermediate, map task assignment + progress, reducer assignment + progress

partition function ex: hash(key) mod R = hopefully uniform distrib. of partitions to nodes

fault tolerance

binomial: 10,000 machines x 0.1% independent failure rate at any moment = 10 P(x >= 1) = 1 - P(x=0) = 1 - C(10000, 0) (0.001)^0 (0.999)^10000 master periodical heartbeart sent to all workers, if no response assumes worker failed if mapper failed: - spawns new map task to new worker and starts over if reducer failed: - new reduce task fetches from mapper again, but only with incomplete partitions (as master knows progress of reducer b4 failure)

EXPENSIVE: assume mappers maintain intermediate key-value data pairs until partitions ocmpletely processed by reducer and saved to disk - assumes RPCs don't fail, otherwise at-least-once

if master fails? restart job Hadoop 2.x replace with resource manager two auto backup to handle and routine checkpoints saving

optimizations (not applicable to project 1) 1:24:30

data locality (assuming distrib. file system) - master assigns splits to workers who already have data stored locally backup tasks (i.e. overwhelming # of "the"'s) - one-last reducer: 99% for a week processing "the" (due to key-skew) - **speculative execution:** spawns a backup straggler to run it in parallel, whichever one finishes first kills the other $$$$ Partitioning Functions - basic version: partition w/ subset of unique keys (hashed) with all their occurrences –> 1 reducer - ![[Pasted image 20241022105020.png]] - custom version: specific partition alg - i.e. to keep all urls of same domain (i.e. ucla.edu) in same partition - i.e. to solve key skew (too many 'the') by distrib. to diff partitions/reducers - manual list of common words (bc not know in advance which one) i.e. ["the", "a"] - by adding uniform distrib. of rand nums to "the1", "the2", "the3" - ANOTHER map-reduce after this to combine "the1"+"the2"+"the3" etc. $$ 2 Ordering Guarantees 3 Combiner 4 Input and Output 5 Side Effects 6 Skipping Bad Records 7 Local Execution 8 Status Information 9 Counters