Distributed computing lecture notes pdf

Parallel and Distributed Systems (CMSC5702) Student/Faculty Expectations on Teaching and Learning Instructor: Prof. John C.S. Lui This course introduces concepts, models, and implementations related to parallel and distributed computing. For parallel computing, we will cover computational models, parallel communication operations, performance and scalability, parallel matrix algorithms, parallel algorithms for solving systems of linear equations, parallel sorting algorithms, parallel graph algorithms, parallel algorithms for dynamic programming, parallel searching algorithms. For distributed computing, if we have time, we will cover remote procedure calls, distributed time and coordination, distributed replications, concurrency control, commitment protocols, distributed checkpointing algorithms, global state and the snapshot algorithm, ...etc. Programming assignments will be given so that students can understand the topics discussed in class. Final Examination Final exmination schedule: April 24th, 2015, 7:00 pm to 9:00 pm, Wu Ho Man Yuen Building (WMY) Room 303. Deadline for homework and projet: May 1, 2015, 11:30 pm. Grading Homework: 20% Programming Project: 30% Final Exam: 50% (note: Students need to get at least 25% in the final exam to pass the course) Teaching Assistant: Mr. Mingshen Sun Email: please refer to the first lecture note Related textbooks Introduction to Parallel Computing by V. Kumar, A. Gupta, G. Karypis, V. Kumar Distributed Systems: Concepts and Design by G. Coulouris, J. Dollimore and T. Kindberg (Addison Wesley Press) Parallel and Distributed Computation:Numerical Methods by Bertsekas P. Dimitri; John N. Tsitsiklis Lecture 1 (Introduction and Models of Parallel Computers) Lecture 2 (Basic Communication Operations) Lecture 3 (Performance and Scalability of Parallel Systems) Lecture 4 (Dense Matrix Algorithms) Lecture 5 (Various Matrix Algorithms) Lecture 6 (Parallel Graphs Algorithms) Lecture 7 (Parallel Dynamic Programming) Lecture 8 (Synchronization in Distributed Systems) Lecture 9 (Synchronization in Distributed Systems II) Lecture 10 (Snapshot Algorithm) Lecture 11 (Replication, Consistency and Requests Ordering) Lecture 12 (Fault Tolerance and Distributed Recovery) Lecture 13 (Distributed Transaction Management) Lecture on MapReduce/Hadoop (MapReduce/Hadoop Programming: I) Lecture on MapReduce/Hadoop (MapReduce/Hadoop Programming: II) Homework Due: May 1, 2015, 11:30 pm Programming Project (password protected) Programming Project Due: May 1, 2015, 11:30 pm Recommended Reading Title: Structured Parallel Programming: Patterns for Efficient Computation, by Michael McCool, James Reinders Parallel Programming: for Multicore and Cluster Systems, by Thomas Rauber, Gudula Rünger Parallel Programming with Python, by Jan Palach High Performnace Computing at Amazon "As a cell design becomes more complex and interconnected a critical point is reached where a more integrated cellular organization emerges, and vertically generated novelty can and does assume greater importance." Carl Woese Professor of Microbology, University of Illinois "At the highest level, we're looking at 'scaling out' (vs. 'scaling up,' as in frequency), with multicore architecture. Basically, instead of having one big x86 processor, you could have 16, 32, 64, and so on, up to maybe 256 small x86 processors on one die. We'll have the transistor count (thanks to Moore's Law) to do incredible things we could only dream about a few years ago." "In future architectures, we'll have multiple devices working in parallel. That means you can break a problem up to be solved in pieces. In these new architectures, the biggest problem won't be building the hardware. The biggest challenge will be helping people configure algorithms to solve problems in parallel, and providing the software tools to extract that inherent parallelism out of the programs." Stephen S. Pawlowski Chief Technology Officer of Intel's Digital Enterprise Group Resources: Page 2 Note: These lecture notes were slightly modified from the ones posted on the 6.824 course website from Spring 2015. Distributed systems What is a distributed system? multiple networked cooperating computers Example: Internet E-Mail, Athena file server, Google MapReduce, Dropbox, etc. Why distribute? to connect physically separate entities to achieve security via physical isolation to tolerate faults via replication at separate sites to increase performance via parallel CPUs/mem/disk/net ...but: complex, hard to debug new classes of problems, e.g. partial failure (did he accept my e-mail?) Leslie Lamport: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." Advice: don't distribute if a central system will work Why take this course? interesting -- hard problems, non-obvious solutions active research area -- lots of progress + big unsolved problems used by real systems -- unlike 10 years ago -- driven by the rise of big Web sites hands-on -- you'll build a real system in the labs Course structure See the course website. Course components Lectures about big ideas, papers, labs Readings: research papers as case studies please read papers before class paper for today: MapReduce paper each paper has a question for you to answer and one for you to ask (see web site) submit question & answer before class, one or two paragraphs Mid-term quiz in class, and final exam Labs: build increasingly sophisticated fault-tolerant services First lab is due on Monday Project: design and build a distributed system of your choice or the system we pose in the last month of the course teams of two or three project meetings with course staff demo in last class meeting Main topics Example: a shared file system, so users can cooperate, like Dropbox but this lecture isn't about dropbox specifically just an example goal to get feel for distributed system problems lots of client computers Architecture Choice of interfaces Monolithic file server? Block server(s) -> FS logic in clients? Separate naming + file servers? Separate FS + block servers? Single machine room or unified wide area system? Wide-area dramatically more difficult. Client/server or peer-to-peer? Interact w/ performance, security, fault behavior. Implementation How do clients/servers communicate? Direct network communication is pretty painful Want to hide network stuff from application logic Most systems organize distribution with some structuring framework(s) RPC, RMI, DSM, MapReduce, etc. Performance Distribution can hurt: network b/w and latency bottlenecks Lots of tricks, e.g. caching, threaded servers Distribution can help: parallelism, pick server near client Idea: scalable design We would like performance to scale linearly with the addition of machines N x servers -> N x total performance Need a way to divide the load by N divide the state by N split by user split by file name "sharding" or "partitioning" Rarely perfect -> only scales so far Global operations, e.g. search Load imbalance One very active user One very popular file -> one server 100%, added servers mostly idle -> N x servers -> 1 x performance Fault tolerance Dropbox: ~10,000 servers; some fail Can I use my files if there's a failure? Some part of network, some set of servers Maybe: replicate the data on multiple servers Perhaps client sends every operation to both Maybe only needs to wait for one reply Opportunity: operate from two "replicas" independently if partitioned? Opportunity: can 2 servers yield 2x availability AND 2x performance? Consistency Contract w/ apps/users about meaning of operations e.g. "read yields most recently written value" hard due to partial failure, replication/caching, concurrency Problem: keep replicas identical If one is down, it will miss operations Must be brought up to date after reboot If net is broken, both replicas maybe live, and see different ops Delete file, still visible via other replica "split brain" -- usually bad Problem: clients may see updates in different orders Due to caching or replication I make grades.txt unreadable, then TA writes grades to it What if the operations run in different order on different replicas? Consistency often hurts performance (communication, blocking) Many systems cut corners -- "relaxed consistency" Shifts burden to applications Labs Focus: fault tolerance and consistency -- central to distributed systems. lab 1: MapReduce labs 2/3/4: storage servers progressively more sophisticated (tolerate more kinds of faults) progressively harder too! patterned after real systems, e.g. MongoDB Lab 4 has core of a real-world design for 1000s of servers What you'll learn from the labs: easy to listen to lecture / read paper and think you understand building forces you to really understand "I hear and I forget, I see and I remember, I do and I understand" (Confucius?) you'll have to do some design yourself we supply skeleton, requirements, and tests but we leave you substantial scope to solve problems your own way you'll get experience debugging distributed systems Test cases simulate failure scenarios: distributed systems are tricky to debug: concurrency and failures many client and servers operating in parallel test cases make servers fail at the "most" inopportune time think first before starting to code! otherwise your solution will be a mess and/or, it will take you a lot of time code review learn from others judge other solutions We've tried to ensure that the hard problems have to do w/ distributed systems: not e.g. fighting against language, libraries, etc. thus Go (type-safe, garbage collected, slick RPC library) thus fairly simple services (MapReduce, key/value store) Lab 1: MapReduce help you get up to speed on Go and distributed programming first exposure to some fault tolerance motivation for better fault tolerance in later labs motivating app for many papers popular distributed programming framework many descendants frameworks Computational model aimed at document processing split doc -> K1 k, list values run Map(K1 key, list values) on each split -> list kvps run Reduce(K2 key, list values) on each partition -> list merge result write a map function and reduce function framework takes care of parallelism, distribution, and fault tolerance some computations are not targeted, such as: anything that updates a document Example: wc word count In Go's implementation, we have: func Map(value string) *list.List the input is a split of the file wc is called on a split is just a partion of the file, as decided by MapReduce's splitter (can be customized, etc.) returns a list of key-value pairs the key is the word (like 'pen') the value is 1 (to indicate 'pen' occurred once) Note: there will be multiple entries in the list if 'pen' shows up more times func Reduce(key string, values *list.List) string the input is a key and a list of (all?) the values mapped to that key in the Map() phase so here, we would expect a Reduce('pen', [1,1,1,1]) call if pen appeared 4 times in the input file TODO: not clear if it's also possible to get three reduce calls as follows: Reduce('pen', [1,1]) -> 2 + Reduce('pen', [1,1]) -> 2 Reduce('pen', [2,2]) the paper seems to indicate Reduce's return value is just a list of values and so it seems that the association of those values with the key 'pen' in this case would be lost, which would prevent the 3rd Reduce('pen') call Example: grep map phase master splits input in M partitions calls Map on each partition map(partition) -> list(k1,v1) search partition for word produce a list with one item if word shows up, nil if not partition results among R reducers reduce phase Reduce job collects 1/R output from each Map job all map jobs have completed! reduce(k1, v1) -> v2 identity function: v1 in, v1 out merge phase Performance number of jobs: M x R map jobs how much speed up do we get on N machines? ideally: N bottlenecks: stragglers network calls to collect a Reduce partition network calls to interact with FS disk I/O calls Fault tolerance model master is not fault tolerant assumption: this single machine won't fail during running a MapReduce app but many workers, so have to handle their failures assumption: workers are fail stop they fail and stop (e.g., don't send garbled weird packets after a failure) they may reboot What kinds of faults might we want to tolerate? network: lost packets duplicated packets temporary network failure server disconnected network partitioned server: server crash+restart (master versus worker?) server fails permanently (master versus worker?) all servers fail simultaneously -- power/earthquake bad case: crash mid-way through complex operation what happens if we fail in the middle of map or reduce? bugs -- but not in this course what happens when bug in map or reduce? same bug in Map over and over? management software kills app malice -- but not in this course retry -- e.g. if packet is lost, or server crash+restart packets (TCP) and MapReduce jobs may execute MapReduce job twice: must account for this replicate -- e.g. if one server or part of net has failed replace -- for long-term health Retry jobs network falure: oops execute job twice ok for MapReduce, because map()/reduce() produces same output map()/reduce() are "functional" or "deterministic" how about intermediate files? worker failure: may have executed job or not so, we may execute job more than once! but ok for MapReduce as long as map() and reduce() functions are deterministic what would make map() or reduce() not deterministic? is executing a request twice in general ok? no. in fact, often not. unhappy customer if you execute one credit card transaction several times adding servers easy in MapReduce -- just tell master hard in general server may have lost state (need to get new state) server may have rebooted quickly may need to recognize that to bring server up to date server may have a new role after reboot (e.g., not the primary) these harder issues you would have to deal with to make the MapReduce master fault tolerant topic of later labs Lab 1 code The lab 1 app (see main/wc.go): stubs for map() and reduce() you fill them out to implement word count (wc) how would you write grep? The lab 1 sequential implementation (see mapreduce/mapreduce.go): demo: run wc.go code walk through start with RunSingle() The lab 1 worker (see mapreduce/worker.go): the remote procedure calls (RPCs) arguments and replies (see mapreduce/common.go). Server side of RPC RPC handlers have a particular signature RunWorker rpcs.Register: register named handlers -- so Call() can find them Listen: create socket on which to listen for RPC requests for distributed implementation, replace "unix" w. "tcp" replace "me" with a tuple name ServeConn: runs in a separate thread (why?) serve RPC concurrently a RPC may block Client side of RPC call() (see common.go) make an RPC lab code dials for each request typical code uses a network connection for several requests but, real must be prepared to redial anyway a network connection failure, doesn't imply a server failure! we also do this to introduce failure scenarios easily intermittent network failures just loosing the reply, but not the request The lab 1 master (see mapreduce/master.go) You write it You will have to deal with distributing jobs You will have to deal with worker failures results matching "" No results matching "" distributed computing lecture notes pdf. parallel and distributed computing lecture notes. parallel and distributed computing lecture notes pdf. distributed and cloud computing lecture notes