CS 7210 Distributed Computing Reading Notes

Jie Wu, Jack

Spring 2023

1 Lesson #1 What Good are Models and What Models are Good

1.1 The Publication Details

Author: Fred B. Schneider

Title: What Good are Models and What Models are Good?

Journal: N.A.

DOI: `http://www.cs.cornell.edu/courses/cs5414/2012fa/publications/S93.pdf`

1.2 Summary

key attributes of a distributed system: pp.2, the two key attributes of a distributed system are

- process execution speeds and message delivery delays
- failure modes

tractability and the art of attribute selection: pp.2, defining an accurate model is not difficult, defining an accurate and tractable model is, and accurate and tractable model will include exactly those attributes that affect the phenomena of interest, selecting these attributes requires taste and insight, level of detail is a key issue

model selection criteria: pp.2, in building models for distributed systems we typically seek answers to two fundamental questions

feasibility: what classes of problems can be solved

cost: for those classes that can be solved, how expensive must the solution be

asynchronous vs. synchronous: pp.4, because all systems are asynchronous, a protocol designed for use in an asynchronous system can be used in any distributed system, this is a compelling argument for studying asynchronous systems
pp.4, in theory any system that employs reasonable schedulers can be regarded as being synchronous, because there are then (trivial) bounds on the relative speeds of processes and channels, protocols that assume the system is synchronous exhibit performance degradation as the ratios of the various process speeds and delivery delays increase, reasonable throughput can be attained by these protocols only when processes execute at about the same speed and delivery delays are not too large

synchronous and time-based protocol: pp.4, asserting that the relative speeds of processes is bounded is equivalent to assuming that all processors in the system have access to approximately rate-synchronized real-time clocks, this is because either one can be used to implement the other, thus timeouts and other time-based protocol techniques are possible only when a system is synchronous

fault tolerance measure: pp.5, it is faulty components that we count not occurrences of faulty behavior, by way of contrast in classical work on fault-tolerant computing system it is the occurrences of faulty behavior that are counted, a system is t-fault tolerant when that system will continue satisfying its specification provided that no more than t of its components are faulty, once t has been chosen it is not difficult to derive the more traditional statistical measures of reliability, we simply compute the probabilities of having various configurations of 0 through t faulty components, so no expressive power is lost by counting faulty components rather than counting fault occurrences
pp.5, since replication is the only way to tolerate faulty components, the architecture and cost of implementing a t-fault tolerant system very much depends on exactly how fault occurrences are being attributed to components, incorrect attribution leads to an inaccurate distributed system model

failure model: pp.6, failure models commonly found in the distributed systems literature include

failstop: a processor fails by halting, the fact that a processor has failed is detectable by other processors

crash: a processor fails by halting, the fact that a processor has failed may not be detectable by other processors

crash + link: a link fails by losing some messages but does not delay, duplicate, or corrupt messages

receive-omission: a processor fails by receiving only a subset of the messages

send-omission: a processor fails by transmitting only a subset of the messages

general omission: receive-omission and/or send-omission

Byzantine failure: a processor fails by exhibiting arbitrary behavior

failstop failures are the least disruptive, crash failures in asynchronous systems are harder to deal with than failstop failures, in synchronous systems however the crash and failstop models are equivalent, the next four failure models all deal with message loss, Byzantine failures are the most disruptive, a system that can tolerate Byzantine failures can tolerate anything
pp.6, a reasonable abstraction for a processor in a distributed system is an object that sends and receives messages, the failure models given above concern ways that such an abstraction might be faulty, failure models involving the contents of memory or the functioning of an ALU for example concern internal (and largely irrelevant) details of the processor abstraction

model selection: pp.8, a system designed assuming that Byzantine failures are possible can tolerate anything, assuming Byzantine failures is prudent in mission critical systems, for most applications however it suffices to assume a more benign failure model

1.3 Key Facts & Ideas

special message of synchronous system: pp.5, by restricting consideration to synchronous systems, the act of not sending a message can convey information to processes

fault tolerance and distributed system: pp.7, all methods for achieving fault tolerance employ replication of function using components that fail independently, in a distributed system the physical separation and isolation of processors linked by a communications network ensures that components fail independently, hence implementing a distributed system is the only way to achieve fault tolerance

replication for failure: pp.7, failures can be detected only by replicating actions in failure-independent ways, if the results of performing a set of replicated actions disagree then a failure has occurred

replication in space: perform the action using components that are physically and electrically isolated

replication in time: let a single device repeatedly perform the action

pp.7, for Byzantine failures $t+1$-fold replication permits t-fault tolerant failure detection but not masking, in order to implement t-fault tolerant masking $2t+1$-fold replication is needed, since then as many as t values can be faulty without causing the majority value to be faulty
for failstop failures a single component suffices for detection, and $t+1$-fold replication is sufficient for masking the failure of as many as t faulty components

model vs. reality: pp.8, the various models can and should be regarded as limiting cases, the behavior of a real system is bounded by our models

2 Lesson #1 Fallacies of Distributed Systems

2.1 The Publication Details

Author:

Title: Fallacies of Distributed Computing Explained

Journal: N.A.

DOI: `https://arnon.me/wp-content/uploads/Files/fallacies.pdf`

2.2 Summary

the eight fallacies are

the network is reliable: you need to think about hardware and software redundancy, on the software side you need to think about messages/calls getting lost whenever you send a message/make a call over the wire

latency is zero: taking latency into consideration means you should strive to make as few as possible calls and you'd want to move as much data out in each of this calls

bandwidth is infinite: while acting on the realization the latency is not zero we modeled few large messages, bandwidth limitations direct us to strive to limit the size of the information we send over the wire

the network is secure: security is a system quality attribute that needs to be taken into consideration starting from the architectural level, security is usually a multi-layered solution that is handled on the network, infrastructure, and application levels, in essence you need to perform threat modeling to evaluate the security risks, then following further analyses decide which risk are and should be mitigated by what measures

topology doesn't change: try not to depend on specific endpoints or routes, either provide location transparency or provide discovery services
another strategy is to abstract the physical structure of the network, the most obvious example for this is DNS names instead of IP addresses

there is one administrator: you need to think about multiple administrators for pinpointing a problem and/or upgrades
the administrators will most likely not be part of your development team, so we need provide them with tools to diagnose and find problems

transport cost is zero: one interpretation is that going from the application level to the transport level is not free because we have to do marshaling to get data unto the wire, which takes both computer resources and adds to the latency
another interpretation is that the costs for setting and running the network are not free

the network is homogeneous: you have to assume interoperability will be needed sooner or later and be ready to support it from day one
do not rely on proprietary protocols, do use standard technologies that are widely accepted

3 Lesson #3 Logical Time in Distributed Systems

3.1 The Publication Details

Author: M. Raynal and M. Singhal

Title: Logical Time: A Way to Capture Causality in Distributed Systems

Journal: IRISA Technical Report

DOI: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=047b65692819dd712f976a91`

3.2 Summary

pp.18, there is no built-in physical time in distributed systems and it is only possible to realize an approximation of it; pp.1, the logical time, which advances in jumps, is sufficient to capture the fundamental monotonicity property associated with causality in distributed system, this paper reviews three ways to define logical time (scalar time, vector time, matrix time) that have been proposed to capture causality between events of a distributed computation

3.2.1 Scalar time

definition: pp.8, the logical local clock of a process p_i and its local view of the global time are squashed into one integer variable C_i, rules R1 and R2 to update the clocks are as follows

R1: before executing an event process p_i executes the following: $C_i := C_i + d$, each time R1 is executed d can have a different value

R2: each message piggybacks the clock value of its sender at sending time, when a process p_i receives a message with timestamp C_{msg} it executes the following:

1. $C_i := \max(C_i, C_{msg})$
2. execute R1
3. deliver the message

property: pp.8, scalar clocks can be used to totally order the events in a distributed system as follows: the timestamp of an event is denoted by a tuple (t, i) where t is its time of occurrence and i is the identity of the process where it occurred, the total order relation $\prec$ on two events x and y with timestamps (h, i) and (k, j) respectively is defined as follows

$$x \prec y \iff (h < k \text{ or } (h = k \text{ and } i < j))$$

pp.9, if the increment value d is always 1 the scalar time has the following interesting property: if event e has a timestamp h then $h - 1$ represents the minimum logical duration, counted in units of events, required before producing the event e, we call it the **height** of the event e, in short *height(e)*, in other words $h - 1$ events have been produced sequentially before the event e regardless of the processes that produced these events
pp.9, however system of scalar clocks is not strongly consistent, that is for two events e_1 and e_2, $C(e_1) < C(e_2) \not\Rightarrow e_1 \rightarrow e_2$, reason for this is that the logical local clock and logical global clock of a process are squashed into one resulting in the loss of information

3.2.2 Vector time

definition: pp.9, in the system of vector clocks the time domain is represented by a set of n-dimensional non-negative integer vectors, each process p_i maintains a vector $vt_i[1..n]$ where $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process pi, $vt_i[j]$ represents process p_i's latest knowledge of process p_j local time, if $vt_i[j] = x$ then process p_i knows that local time at process p_j has progressed till x, the entire vector vt_i constitutes p_i's view of the logical global time
pp.10, process p_i uses the following two rules to update its clock

R1: before executing an event it updates its local logical time as follows

$$vt_i[i] := vt_i[i] + d \qquad (d > 0)$$

R2: each message m is piggybacked with the vector clock vt of the sender process at sending time, on the receipt of such a message (m, vt) process p_i executes the following sequence of actions

1. $vt_i[k] := \max(vt_i[k], vt[k])\ \forall 1 \leq k \leq n$
2. execute R1
3. deliver the message m

the timestamp associated with an event is the value of the vector clock of its process when the event is executed

property: pp.10–11, there are three

isomorphism: the following three relations are defined to compare two vector timestamps vh and vk

$$vh \leq vk \iff \forall x : vh[x] \leq vk[x]$$
$$vh < vk \iff vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$
$$vh \parallel vk \iff \text{not } (vh < vk) \text{ and not } (vk < vh)$$

pp.10, if two events x and y have timestamps vh and vk respectively, then

$$x \rightarrow y \iff vh < vk$$
$$x \parallel y \iff vh \parallel vk$$

pp.11, thus there is an isomorphism between the set of partially ordered event produced by a distributed computation and their timestamps, if processes of occurrence of events are known, the test to compare two timestamps can be simplified as follows: if events x and y respectively occurred at process p_i and p_j and are assigned timestamps (vh, i) and (vk, j) respectively, then

$$x \rightarrow y \iff vh[i] < vk[i]$$
$$x \parallel y \iff vh[i] > vk[i] \text{ and } vh[j] < vk[j]$$

strong consistency: pp.11, the system of vector clocks is strongly consistent, thus by examining the vector timestamp of two events we can determine if they are causally related, however the dimension of vector clocks cannot be less than n for this property

event counting: pp.11, if d is always 1 in the rule R1, then the ith component of vector clock at process p_i, $vt_i[i]$, denotes the number of events that have occurred at p_i until that instant, e.g. if an event e has timestamp vh then $vh[j]$ denotes the number of events executed by process p_j that causally precede e and $\sum_j vh[j] - 1$ represents the total number of events that causally precede e in the distributed computation

application: pp.12, since vector time tracks causal dependencies exactly, it finds a wide variety of applications, e.g. distributed debugging, causal ordering communication, causal distributed shared memory, establishing global breakpoints, determining the consistency of checkpoints

3.2.3 Matrix time

description: pp.12, in a system of matrix clocks the time is represented by a set of $n \times n$ matrices of non-negative integers, a process p_i maintains a matrix $mt_i[1..n, 1..n]$ where

- $mt_i[i, i]$ denotes the local logical clock of p_i and tracks the progress of the computation at process p_i
- $mt_i[k, l]$ denotes the knowledge that process p_i has about the knowledge of p_k about the logical local clock of p_l, the entire matrix mt_i denotes p_i's local view of the logical global time

note that row $mt_i[i, .]$ is nothing but the vector clock $vt_i[.]$ and exhibits all properties of vector clocks
pp.12, process p_i uses the following rules to update its clock

R1: before executing an event it updates its local logical time as follows

$$mt_i[i, i] := mt_i[i, i] + d \qquad (d > 0)$$

R2: each message m is piggybacked with matrix time mt, when p_i receives such a message (m, mt) from a process p_j, p_i executes the following sequence of actions

1. $mt_i[i, k] := \max(mt_i[i, k], mt[j, k])\ \forall 1 \leq k \leq n$
   $mt_i[k, l] := \max(mt_i[k, l], mt[k, l])\ \forall 1 \leq k, l \leq n$
2. execute R1
3. deliver message m

property: pp.13, clearly vector $mt_i[i, .]$ contains all the properties of vector clocks, in addition matrix clocks have the following property: $\min_k(mt_i[k, l] \geq t)$ implies that process p_i knows that every other process p_k knows that p_l's local time has progressed till t, in many applications this implies that processes will no longer require from p_l certain information and can use this fact to discard

obsolete information
pp.13, if d is always 1 in the rule R1, then $mt_i[k, l]$ denotes the number of events occurred at p_l and known by p_k as far as p_i's knowledge is concerned

3.2.4 Implementation

pp.13, it has been shown that if vector clocks have to satisfy the strong consistency property, then in general vector timestamps must be at least of size n, therefore in general the size of vector timestamp is the number of processes involved in a distributed computation, however several optimizations are possible

Singhal-Kshemkalyani's differential technique: pp.13, based on the observation that between successive events at a process, only few entries of the vector clock at that process are likely to change, this is more true when the number of processes is large because only few of them will interact frequently by passing messages, when a process p_i sends a message to a process p_j, it piggybacks only those entries of its vector clock that differ since the last message send to p_j
pp.14, this technique can substantially reduce the cost of maintaining vector clocks in large systems if process interaction exhibits temporal or spatial localities, however it requires that communication channels be FIFO, and a process needs to maintain two additional vectors to store the information regarding the clock values when the last interaction was done with other processes

Fowler-Zwaenepoel's direct-dependency technique: pp.15, in this technique a process only maintains information regarding *direct* dependencies on other processes, a vector time for an event that represents *transitive* dependencies on other processes is constructed off-line from a recursive search of the direct dependency information at processes, in this technique a process p_i maintains a dependency vector D_i that is initially $D_i[j] = 0$ for $j = 1..n$ and is updated as follows

- whenever an event occurs at p_i, $D_i[i] := D_i[i] + 1$
- when a process p_j sends a message m to p_i, it piggybacks the updated value of $D_j[j]$ in the message
- when p_i receives a message from p_j with piggybacked value d, p_i updates its dependency vector as follows: $D_i[j] := \max(D_i[j], d)$

at any instance $D_i[j]$ denotes the sequence number of the latest event on process p_j that directly affects the current state
pp.15, this technique results in considerable saving in the cost as only one scalar is piggybacked on every message, however the dependency vector does not represent transitive dependencies, the transitive dependency (or the vector timestamp) of an event is obtained by recursively tracing the direct-dependency vectors of processes, clearly this will have overhead and will involve latencies, therefore this technique is not suitable for applications that require on-the-fly computation of vector timestamps, nonetheless this technique is ideal for applications where computation of causal dependencies is performed off-line, e.g. causal breakpoint and asynchronous checkpointing recovery

Jard-Jourdan's adaptive technique: pp.15, in this technique a process must observe an event (i.e. update and record its dependency vector) after receiving a message but before sending out any message, if events occur very frequently this technique will require recording the history of a large number of events
pp.16, Jard-Jordan define a *pseudo-direct* relation $\ll$ on the events of a distributed computation as follows: if events e_i and e_j happen at processes p_i and p_j respectively, then $e_j \ll e_i$ iff there

exists a path of message transfers that starts after e_j on p_j and ends before e_i on p_i such that there is no observed event on the path, the partial vector clock p_vt_i at process p_i is a list of tuples of the form (j, v) indicating that the current state of p_i is pseudo-dependent on the event on process p_j whose sequence number is v, initially at a process p_i $p\_vt_i = \{(i, 0)\}$; pp.18, considering the timestamp of an event the set of the observed events that are its predecessors can be easily computed

3.3 Key Facts & Ideas

abstraction of process: pp.3, the actions of a process are modeled as three types of events—internal events, message send events, and message receive events, send and receive events establish causal dependency from the sender process to the receiver process

implication of causality: pp.3, the knowledge of the causal precedence relation among processes helps to solve

- **distributed algorithm design:** mutual exclusion algorithms, database replication, deadlock detection algorithms
- **tracking of dependent events:** build checkpoints for failure recovery, detect file inconsistencies in network partitioning
- **knowledge about the progress:** discard obsolete information, garbage collection, termination detection
- **concurrency measure:** all events that are not causally related can be executed concurrently

symbol: pp.5, if $e_1 \rightarrow e_2$ then event e_2 is directly or transitively dependent on event e_1, if $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$ then event e_1 and e_2 are said to be concurrent and are denoted as $e_1 \parallel e_2$, clearly for any two events e_1 and e_2 in a distributed execution $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$ or $e_1 \parallel e_2$

observation level: pp.6, an observation level defines a projection of the events in the distributed computation, more specifically it defines a set of relevant events R and the restriction of $\rightarrow$ to the events in R denoted by $\rightarrow_R$

definition of logical clock: pp.6, a system of logical clocks consists of a time domain T and a logical clock C, elements of T form a partially ordered set over a relation $<$, this relation is usually called happened before or causal precedence, the logical clock C is a function that maps an event e in a distributed system to an element in the time domain T denoted as $C(e)$ and called the **timesstamp** of e and is defined as follows

$$C : H \rightarrow T$$

such that the following property is satisfied

$$e_1 \rightarrow e_2 \implies C(e_1) < C(e_2)$$

this monotonicity property is called the **clock consistency condition**, when T and C satisfy the following condition

$$e_1 \rightarrow e_2 \Longleftrightarrow C(e_1) < C(e_2)$$

the system of clocks is said to be **strongly consistent**

implementation of logical clock: pp.7, implementation of logical clocks requires addressing two issues

data structure: local to every process to allow two capabilities: (1) local logical clock (2) logical global clock

protocol: set of rules to update the data structure to ensure the consistency condition, consists of the following two rules

R1: governs how the local logical clock is updated by a process when it executes an event

R2: governs how a process update its view of the global time and global progress, dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time

4 Lesson #4 Consistent Global States of Distributed Systems

4.1 The Publication Details

Author: Özalp Babaoglŭ Keith Marzullo

Title: Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms

Journal: N.A.

DOI: `http://www.cs.cornell.edu/courses/cs5414/2012fa/publications/BM93.pdf`

4.2 Summary

pp.1, distributed systems are adequately modeled as asynchronous systems, in this paper we consider global predicate evaluation as a canonical problem in order to survey concepts and mechanisms that are useful in coping with uncertainty in distributed computations, we illustrate the utility of the developed techniques by examining distributed deadlock detection and distributed debugging as two instances of global predicate evaluation; pp.2, the goal of global predicate evaluation (GPE) is to determine whether the global state of the system satisfies some predicate Φ

4.3 Global States, Cuts and Runs

pp.5, let σ_i^k denote the local state of process p_i immediately after having executed event e_i^k and let σ_i^0 be its initial state before any events are executed, the **global state** of a distributed computation is an n-tuple of local state $\Sigma = (\sigma_1, \ldots, \sigma_n)$ one for each process, a **cut** of a distribute computation is a subset C of its global history H and contains an initial prefix of each of the local histories, we can specify such a cut $C = h_1^{c_1} \cup \cdots \cup h_n^{c_n}$ through the tuple of natural numbers $(c_1, \ldots, c_n)$ corresponding to the index of the last event included for each process, the set of last events $(e_1^{c_1}, \ldots, e_n^{c_n})$ included in cut $(c_1, \ldots, c_n)$ is called the **frontier** of the cut, clearly each cut defined by $(c_1, \ldots, c_n)$ has a corresponding global state which is $(\sigma_1^{c_1}, \ldots, \sigma_n^{c_n})$; pp.6, a **run** of a distributed computation is total ordering R that includes all of the events in the global history and that is consistent with each local history, a run need not correspond to any possible execution and a single distributed computation may have many runs, each corresponding to a different execution

4.4 Consistency

pp.7, a cut is **consistent** if for all events e and e'

$$(e \in C) \wedge (e' \rightarrow e) \implies e' \in C$$

in other words a consistent cut is left closed under the causal precedence relation, in its graphical representation if all arrows that intersect the cut have their bases to the left and heads to the right of it, then the cut is consistent otherwise it is inconsistent, a **consistent global state** is one corresponding to a consistent cut, just as a scalar time value denotes a particular instant during a sequential computation, the frontier of a consistent cut establishes an instant during a distributed computation, an event e is before/after a cut C if e is to the left/right of the frontier of C
pp.8, predicate values are meaningful only when evaluated in consistent global states since these characterize exactly the states that could have taken place during an execution, a run R is said to be **consistent** if for all events $e \rightarrow e'$ implies that e appears before e' in R, if the run is consistent then the global states in the sequence will all be consistent as well, we will use the term "run" to refer to both the sequence of events and the sequence of resulting global states, each consistent global state Σ^i of the run is obtained from the previous state Σ^{i-1} by some process executing the single event e^i, for two such consistent global states of run R we say that Σ^{i-1} leads to Σ^i in R, let $\rightsquigarrow_R$ denote the transitive closure of the leads-to relation in a given run R, we say that Σ' is **reachable** from Σ in run R if and only if $\Sigma \rightsquigarrow_R \Sigma'$, every global state is reachable from the initial global state Σ^{00}
pp.8, the set of all consistent global states of a computation along with the leads-to relation defines a **lattice**, let $\Sigma^{k_1 \ldots k_n}$ be a shorthand for the global state $(\sigma_1^{k_1}, \ldots, \sigma_n^{k_n})$ and let $k_1 + \cdots + k_n$ be its **level**, a path in the lattice is a sequence of global states of increasing level where the level between any two successive elements differs by one, each such path corresponds to a consistent run of the computation, the run is said to pass through the global states included in the path, note that in an asynchronous distributed system it is impossible to identify the run corresponding to the actual execution of the computation

4.5 Distributed Snapshots

pp.20, we will now develop a snapshot protocol that constructs only consistent global states, for simplicity we will assume that the channels implement FIFO delivery and we will omit details of how individual processes return their local states to p_0, for each channel from p_i to p_j its **channel state** $\chi_{i,j}$ are those messages that p_i has sent to p_j but p_j has not yet received, channel state is convenient for e.g. constructing the waits-for graph in deadlock detection: process p_i is blocked on p_j if σ_i records the fact that there is an outstanding request to p_j and $\chi_{j,i}$ contains no response messages, let IN_i be the set of processes that have channels connecting them directly to p_i and OUT_i be the set of processes to which p_i has a channel, channels from $p_j \in IN_i$ to p_i are called incoming while channels from p_i to $p_j \in OUT_i$ are called outgoing with respect to p_i, for each execution of the snapshot protocol a process p_i will record its local state σ_i and the states of its incoming channels $\chi_{j,i}$ for all $p_j \in IN_i$

4.5.1 Snapshot protocols

The Chandy-Lamport distributed snapshot protocol takes the following steps for a snapshot

1. process p_0 starts the protocol by sending itself a "take snapshot" message
2. let p_f be the process from which p_i receives the "take snapshot" message for the first time, upon receiving this message p_i records its local state σ_i and relays the "take snapshot" message along all of its outgoing channels, no intervening events on behalf of the underlying computation are executed between these steps, channel state $\chi_{f,i}$ is set to empty and p_i starts recording messages received over each of its other incoming channels
3. let p_s be the process from which p_i receives the "take snapshot" message beyond the first time, process p_i stops recording messages along the channel from p_s and declares channel state $\chi_{s,i}$ as those messages that have been recorded

pp.23, since a "take snapshot" message is relayed only upon the first receipt and since the network is strongly connected, a "take snapshot" message traverses each channel exactly once, when process p_i has received a "take snapshot" message from all of its incoming channels, its contribution to the global state is complete and its participation in the snapshot protocol ends

4.5.2 Properties of snapshots

pp.24, let Σ^a be the global state in which the snapshot protocol is initiated, Σ^f be the global state in which the protocol terminates and Σ^s be the global state constructed, we will show that there exists a run R such that $\Sigma^a \rightsquigarrow_R \Sigma^s \rightsquigarrow_R \Sigma^f$, such a run R can be constructed as follows

1. let r be the actual run the system followed while executing the snapshot protocol and let e^* denote the event when p_i receives "take snapshot" for the first time causing p_i to record its state, an event e_i of p_i is a *prerecording* event if $e_i \rightarrow e_i^*$, otherwise it is a *postrecording* event, consider any two adjacent events $\langle e, e' \rangle$ of r such that e is a postrecording event and e' is a prerecording event, we will show that the order of these two events can be swapped thereby resulting in another consistent run

2. if we continue this process of swapping $\langle \text{postrecording}, \text{prerecording} \rangle$ event pairs, then we will eventually construct a consistent run in which no prerecording event follows a postrecording event, the global state associated with the last prerecording event is therefore reachable from Σ^a and the state Σ^f is reachable from it

3. show that this state is Σ^s, the state that the snapshot protocol constructs

5 Lesson #5 Impossibility of Distributed Consensus with One Faulty Process

5.1 The Publication Details

Author: Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson

Title: Impossibility of Distributed Consensus with One Faulty Process

Journal: Journal of the Association for Computing Machinery, Vol. 32, No. 2, April 1985, pp.374–82

DOI: `http://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf`

5.2 Summary

abstract: pp.1, the consensus problem is for the reliable processes to agree on a binary value, in this paper it is shown that every protocol for this problem has the possibility of nontermination even with only one faulty process, by way of contrast solutions are known for the synchronous case—the "Byzantine generals" problem

conclusion: pp.2, in this paper we show the surprising result that no completely asynchronous consensus protocol can tolerate even a single unannounced process death, we do not consider Byzantine failures and we assume that the message system is reliable—it delivers all messages correctly and exactly once, even with these assumptions the stopping of a single process at an inopportune time can cause any distributed commit protocol to fail to reach agreement; pp.8, it points up the need for more refined models of distributed computing that better reflect realistic assumptions

about processor and communication timings, and for less stringent requirements on the solution to such problems

assumption: pp.2–3, they are

- we do not consider Byzantine failures
- we assume that the message system is reliable—it delivers all messages correctly and exactly once
- crucial to our proof is that processing is completely asynchronous—we make no assumptions about the relative speeds of processes or about the delay time in delivering a message
- we also assume that processes do not have access to synchronized clocks, so algorithms based on time-outs for example cannot be used
- we do not postulate the ability to detect the death of a process, so it is impossible for one process to tell whether another has died or is just running very slowly
- for the purpose of the impossibility proof we require only that some process eventually make a decision
- an "atomic broadcast" capability is assumed, so a process can send the same message in one step to all other processes with the knowledge that if any nonfaulty process receives the message then all the nonfaulty processes will
- every message is eventually delivered as long as the destination process makes infinitely many attempts to receive

model: pp.2, our system model is rather strong so as to make our impossibility proof as widely applicable as possible, processes are modeled as automata (with possibly infinitely many states) that communicate by means of messages

contribution: two theorems

theorem 1: no consensus protocol is totally correct in spite of one fault

theorem 2: there is a partially correct consensus protocol in which all nonfaulty processes always reach a decision, provided no processes die during its execution and a strict majority of the processes are alive initially

5.3 Definition of Consensus Protocol

consensus protocol: pp.3, a **consensus protocol** P is an asynchronous system of N processes ($N \geq 2$), each process p has a one-bit *input register* x_p, an *output register* y_p with values in $\{b, 0, 1\}$, and an unbounded amount of internal storage, the values in the input and output registers together with the program counter and internal storage comprise the **internal state**, *initial states* prescribe fixed starting values for all but the input register, in particular the output register starts with value b, the states in which the output register has value 0 or 1 are distinguished as being **decision states**

transition function: pp.3, process p acts deterministically according to a **transition function**, the transition function cannot change the value of the output register once the process has reached a decision state, that is the output register is "write-once", the entire system P is specified by the transition functions associated with each of the processes and the initial values of the input registers

message: pp.3, processes communicate by sending each other messages, a **message** is a pair (p, m) where p is the name of the destination process and m is a "message value" from a fixed universe M, the *message system* maintains a multiset called the **message buffer** of messages that have been sent but not yet delivered, it supports two abstract operations

send(p, m)**:** places (p, m) in the message buffer

receive(p)**:** deletes some message (p, m) from the buffer and returns m in which case we say (p, m) is delivered, or returns the special null marker $\emptyset$ and leaves the buffer unchanged

thus the message system acts nondeterministically, subject only to the condition that if receive(p) is performed infinitely many times then every message (p, m) in the message buffer is eventually delivered, in particular the message system is allowed to return $\emptyset$ a finite number of times in response to receive(p)

configuration: pp.3, a *configuration* of the system consists of the internal state of each process together with the contents of the message buffer, an **initial configuration** is one in which each process starts at an initial state and the message buffer is empty, a **step** takes one configuration to another and consists of a primitive step by a single process p, let C be a configuration, the step occurs in two phases

1. receive(p) is performed on the message buffer in C to obtain a value $m \in M \cup \{\emptyset\}$
2. depending on p's internal state in C and on m, p enters a new internal state and sends a finite set of messages to other processes

since processes are deterministic the step is completely determined by the pair $e = (p, m)$ which we call an **event**, $e(C)$ denotes the resulting configuration and we say that e can be *applied* to C, note that the event $(p, \emptyset)$ can always be applied to C so it is always possible for a process to take another step

schedule: pp.4, a **schedule** from C is a finite or infinite sequence σ of events that can be applied in turn starting from C, the associated sequence of steps is called a **run**, if σ is finite we let $\sigma(C)$ denote the resulting configuration which is said to be **reachable** from C, a configuration reachable from some initial configuration is said to be *accessible*, hereafter all configurations mentioned are assumed to be accessible
pp.4, the commutativity property of schedules states the following: suppose that from some configuration C, the schedules σ_1, σ_2 lead to configurations C_1, C_2 respectively, if the sets of processes taking steps in σ_1 and σ_2 respectively are disjoint, the σ_2 can be applied to C_1 and σ_1 can be applied to C_2, and both lead to the same configuration C_3

correct: pp.4, a configuration C has **decision value** ν if some process p is in a decision state with $y_p = \nu$, a consensus protocol is **partially correct** if it satisfies two conditions

- no accessible configuration has more than one decision value
- for each $\nu \in \{0, 1\}$ some accessible configuration has decision value ν

pp.4, a process p is **nonfaulty** in a run provided that it takes infinitely many steps and it is faulty otherwise, a run is **admissible** provided that at most one process is faulty and that all messages sent to nonfaulty processes are eventually received, a run is a **deciding** run provided that some process reaches a decision state in that run, a consensus protocol P is **totally correct** *in spite of one fault* if it is partially correct and every admissible run is a deciding run, our main theorem shows that no consensus protocol is totally correct in spite of one fault, we prove it by

contradiction, the basic idea is to show circumstances under which the protocol remains forever indecisive, this involves two steps

1. show that there is some initial configuration in which the decision is not already predetermined
2. construct an admissible run that avoids ever taking a step that would commit the system to a particular decision

pp.5, let V be the set of decision values of configurations reachable from C, C is *bivalent* if $|V| = 2$, C is *univalent* if $|V| = 1$, by the total correctness of P and the fact that there are always admissible runs (see below), $V \neq \emptyset$; pp.6, any deciding run from a bivalent initial configuration goes to a univalent configuration, so there must be some single step that goes from a bivalent to a univalent configuration, such a step determines the eventual decision value, we show that it is always possible to run the system in a way that avoids such steps leading to an admissible nondeciding run

5.4 Key Facts & Ideas

transaction commit problem: pp.2, the problem is for all the data manager processes that have participated in the processing of a particular transaction to agree on whether to install the transaction's results in the database or to discard them, whatever the decision is made all data managers must make the same decision in order to preserve the consistency of the database

6 Lesson #6 Chain Replication for Supporting High Throughput and Availability

6.1 The Publication Details

Author: Robbert van Renesse, Fred B. Schneider

Title: Chain Replication for Supporting High Throughput and Availability

Journal:

DOI: `https://www.cs.cornell.edu/home/rvr/papers/OSDI04.pdf`

6.2 Summary

target: pp.1 LHS, this paper is concerned with storage systems that sit somewhere between file systems and database systems, in particular we are concerned with storage systems, henceforth called storage services, that

- store *objects*
- support *query* operations to return a value derived from a single object
- support *update* operations to atomically change the state of a single object according to some pre-programmed possibly non-deterministic computation involving the prior state of that object

a file system write is thus a special case of our storage service update which in turn is a special case of a database transaction

model: pp.2 RHS, the client view of an object's state can take two values

$Hist_{objID}$**:** update request sequence

$Pending_{objID}$**:** request set

and the client view of possible object's state transitions are

client request r **arrives (T1):** $Pending_{objID} := Pending_{objID} \cup \{r\}$

client request $r \in Pending_{ojbID}$ **ignored (T2):** $Pending_{objID} := Pending_{objID} - \{r\}$

client request $r \in Pending_{ojbID}$ **processed (T3):** $Pending_{objID} := Pending_{objID} - \{r\}$, reply according options $opts$ in $r = \text{query}(objId, opts)$ or $r = \text{update}(objId, newVal, opts)$ and $Hist_{objID} := Hist_{objID} \cdot r$ for the latter

pp.8 RHS, a set of objects can always be grouped into a single **volume**, itself something that can be considered an object for purposes of chain replication, so a designer has considerable latitude in deciding object size

assumption: pp.4 LHS, in response to detecting the failure of a server that is part of a chain (and by the fail-stop assumption all such failures are detected), the chain is reconfigured to eliminate the failed server, for this purpose we employ a service called the *master* that

- detects failures of servers
- informs each server in the chain of its new predecessor or new successor in the new chain obtained by deleting the failed server
- informs clients which server is the head and which is the tail of the chain

in what follows we assume the master is a single process that never fails, this simplifies the exposition but is not a realistic assumption, our prototype implementation of chain replication actually replicates a master process on multiple hosts, using Paxos [16] to coordinate those replicas so they behave in aggregate like a single process that does not fail

experiment: pp.10 RHS, update throughput decreases to 0 at the time of the server failure and then, once the master deletes the failed server from all chains, throughput is actually better than it was initially, this throughput improvement occurs because the server failure causes some chains to be length 2 (rather than 3), reducing the amount of work involved in performing an update
pp.12 LHS, random placement is inferior because with random placement there are more sets of t servers that together store a copy of a chain, and therefore there is a higher probability of a chain getting lost due to failures, however random placement makes additional opportunities for parallel recovery possible if there are enough servers

comment: pp.13 LHS, chain replication supports high throughput for query and update requests, high availability of data objects, and strong consistency guarantees, because storage services built using chain replication can and do exhibit transient outages but clients cannot distinguish such outages from lost messages, thus the transient outages that chain replication introduces do not expose clients to new failure modes, chain replication represents an interesting balance between what failures it hides from clients and what failures it doesn't
pp.13 LHS, when chain replication is employed high availability of data objects comes from carefully selecting a strategy for placement of volume replicas on servers, random placement of volumes does permit availability to scale with the number of servers if this placement strategy is used in concert with parallel data recovery as introduced for GFS

limitation: pp.13 LHS, our current prototype is intended primarily for use in relatively homogeneous LAN clusters, were our prototype to be deployed in a heterogeneous wide-area setting, then uniform random placement of volume replicas would no longer make sense, instead replica placement would have to depend on access patterns, network proximity, and observed host reliability, protocols to re-order the elements of a chain would likely become crucial in order to control load imbalances

6.3 Key Facts & Ideas

fail-stop: pp.3 LHS, servers are assumed to be fail-stop

- each server halts in response to a failure rather than making erroneous state transitions
- a server's halted state can be detected by the environment

strong consistency: pp.1 RHS, the construction of an application that fronts a storage service is often simplified given *strong consistency guarantees* which assert that

- operations to query and update individual objects are executed in some sequential order
- the effects of update operations are necessarily reflected in results returned by subsequent query operations

strong consistency guarantees are often thought to be in tension with achieving high throughput and high availability, the Google file system (GFS) illustrates this thinking, in fact strong consistency guarantees in a large-scale storage service are not incompatible with high throughput and availability, and the new chain replication approach to coordinating fail-stop servers simultaneously supports high throughput, availability, and strong consistency

chain replication: pp.3 LHS, in chain replication the servers replicating a given object $objID$ are linearly ordered to form a *chain*, the first server in the chain is called the *head*, the last server is called the *tail*, and request processing is implemented by the servers roughly as follows

reply generation: the reply for every request is generated and sent by the tail

query processing: each query request is directed to the tail of the chain and processed there atomically using the replica of $objID$ stored at the tail

update processing: each update request is directed to the head of the chain, the request is processed there atomically using replica of $objID$ at the head, then state changes are forwarded along a reliable FIFO link to the next element of the chain (where it is handled and forwarded), and so on until the request is handled by the tail; pp.4 RHS, because updates are sent between elements of a chain over reliable FIFO links the sequence of updates received by each server is a prefix of those received by its successor

strong consistency thus follows because query requests and update requests are all processed serially at a single server (the tail)
pp.3 RHS, processing a query request involves only a single server, and that means query is a relatively cheap operation, but when an update request is processed computation done at $t-1$ of the t servers does not contribute to producing the reply and arguably is redundant, the redundant servers do increase the fault-tolerance though, note that some redundant computation associated with the $t-1$ servers is avoided in chain replication because the new value is computed once by the head and then forwarded down the chain, so each replica has only to perform a write, this forwarding of state changes also means update can be a non-deterministic operation—the non-deterministic choice is made once by the head

request order invariant: pp.4 RHS–pp.5 LHS,

update propagation invariant: for servers labeled i and j such that $i \leq j$ holds (i.e. i is a predecessor of j in the chain) then $Hist^{j}_{objID} \preceq Hist^{i}_{objID}$

in-process requests invariant: if $i \leq j$ then $Hist^{i}_{objID} = Hist^{j}_{objID} \oplus Sent_i$ where $Sent_i$ is a list of update requests that server i has forwarded to some successor but that might not have been processed by the tail

extending a chain: pp.5 RHS, failed servers are removed from chains, but shorter chains tolerate fewer failures, and object availability ultimately could be compromised if ever there are too many server failures, the solution is to add new servers when chains get short, a new server could in theory be added anywhere in a chain, in practice adding a server T^+ to the very end of a chain seems simplest, for a tail T^+ $Sent_{T^+}$ is initialized to an empty list, and $Hist^{T^+}_{objID}$ is initialized such that $Hist^{T}_{objID} = Hist^{T^+}_{objID} \oplus Sent_T$

chain replication vs. primary/backup: pp.6 LHS, chain replication is a form of primary/backup approach which itself is an instance of the state machine approach to replica management, in the primary/backup approach one server designated the *primary*

- imposes a sequencing on client requests and thereby ensures strong consistency holds
- distributes in sequence to other servers known as *backups* the client requests or resulting updates
- awaits acknowledgments from all non-faulty backups
- after receiving those acknowledgments then sends a reply to the client

if the primary fails one of the backups is promoted into that role
pp.6 LHS, with chain replication the primary's role in sequencing requests is shared by two replicas—the head sequences update requests and the tail extends that sequence by interleaving query requests, this sharing of responsibility not only partitions the sequencing task but also enables lower-latency and lower-overhead processing for query requests, because only a single server (the tail) is involved in processing a query and that processing is never delayed by activity elsewhere in the chain vs. the primary/backup approach in which the primary before responding to a query must await acknowledgments from backups for prior updates
pp.6 RHS, in both chain replication and in the primary/backup approach, update requests must be disseminated to all servers replicating an object or else the replicas will diverge, chain replication does this dissemination serially resulting in higher latency than the primary/backup approach where requests were distributed to backups in parallel
pp.6 RHS, the delay to detect a server failure is by far the dominant cost, and this cost is identical for both chain replication and the primary/backup approach, then what makes the difference is the recovery cost for each approach assuming that a server failure has been detected; pp.7 LHS, the worst case outage for chain replication (tail failure) is never as long as the worst case outage for primary/backup (primary failure), and the best case for chain replication (middle server failure) is shorter than the best case outage for primary/backup (backup failure)

parallel data recovery: pp.11 LHS, to understand the advantages of parallel data recovery, consider a server F that fails and was participating in chains C_1, C_2, ..., C_n, for each chain C_i data recovery requires a source from which the volume data is fetched and a host that will become the new element of chain C_i, given enough processors and no constraints on the placement of volumes, it is easy to ensure that the new elements are all disjoint, and with random placement of volumes it is likely that the sources will be disjoint as well, with disjoint sources and new elements data recovery for chains C_1, C_2, ..., C_n can occur in parallel

6.4 References

Paxos: L. Lamport. The part-time parliament. ACM Transactions on Computer Systems, 16(2):133–169, 1998

7 Lesson #7 A Survey of Rollback-Recovery Protocols in Message-Passing Systems

7.1 The Publication Details

Author: E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson

Title: A Survey of Rollback-Recovery Protocols in Message-Passing Systems

Journal: ACM Computing Surveys, Vol. 34, Issue. 3, September 2002, pp.375—408

DOI: `https://www.cs.utexas.edu/~lorenzo/papers/SurveyFinal.pdf`

7.2 Summary

abstract: pp.1, this survey covers rollback-recovery techniques that do not require special language constructs, in the first part of the survey we classify rollback-recovery protocols into

- **checkpoint-based:** rely solely on checkpointing for system state restoration, checkpointing can be coordinated, uncoordinated, or communication-induced
- **log-based:** combine checkpointing with logging of nondeterministic events encoded in tuples called **determinants**, depending on how determinants are logged log-based protocols can be pessimistic, optimistic, or causal

background: pp.1 LHS, many techniques have been developed to add reliability and high availability to distributed systems, which include

- **transactions:** focus on data-oriented applications
- **group communication:** offers an abstraction of an ideal communication system that simplifies the development of reliable applications
- **rollback recovery:** focus on long-running applications such as scientific computing and telecommunication applications

scope: pp.2 LHS, the coverage excludes the use of rollback recovery in many related fields such as hardware-level instruction retry, distributed shared memory, real-time systems, and debugging, the coverage also excludes the issues of using rollback recovery when failures could include Byzantine modes or are not restricted to the fail-stop model, also excluded are rollback-recovery techniques that rely on special language constructs such as recovery blocks and transactions, finally the section on implementation exposes many relevant issues related to implementing checkpointing on uniprocessors

7.3 Key Facts & Ideas

transparent: pp.2 LHS, a system may rely on the application to decide when and what to save on stable storage, or it may provide the application programmer with linguistic constructs to structure the application, we focus in this survey on *transparent* techniques which do not require any intervention on the part of the application or the programmer, the system automatically takes checkpoints according to some specified policy and recovers automatically from failures if they occur

rollback propagation: pp.2 RHS, message-passing systems complicate rollback recovery because messages induce inter-process dependencies during failure-free operation, these dependencies may force some of the processes that did not fail to roll back creating what is commonly called **rollback propagation**, under some scenarios rollback propagation may extend back to the initial state of the computation losing all the work performed before a failure, this situation is known as the **domino effect**, the domino effect may occur if each process takes its checkpoints independently—an approach known as *independent* or *uncoordinated checkpointing*, two techniques to avoid the domino effect are

- **coordinated checkpointing:** processes coordinate their checkpoints in order to save a system-wide consistent state, this consistent set of checkpoints can then be used to bound rollback propagation
- **communication-induced checkpointing:** forces each process to take checkpoints based on information piggybacked on the application messages received from other processes

checkpoint-based vs. log-based: pp.2 RHS, log-based rollback recovery combines checkpointing with logging of nondeterministic events, log-based rollback recovery relies on the *piecewise deterministic* (PWD) assumption, which postulates that all nondeterministic events that a process executes can be identified and that the information necessary to replay each event during recovery can be logged in the event's **determinant**, log-based rollback recovery in general enables a system to recover beyond the most recent set of consistent checkpoints, it is therefore particularly attractive for applications that frequently interact with the outside world, which consists of all input and output devices that cannot roll back

pp.3 log-based rollback recovery has three flavors depending on how the determinants are logged to stable storage

- **pessimistic logging:** the application has to block waiting for the determinant of each nondeterministic event to be stored on stable storage before the effects of that event can be seen by other processes or the outside world, it simplifies recovery but hurts failure-free performance
- **optimistic logging:** the application does not block, and determinants are spooled to stable storage asynchronously, it reduces the failure-free overhead but complicates recovery
- **causal logging:** low failure-free overhead and simpler recovery are combined by striking a balance between optimistic and pessimistic logging

pp.6 LHS, log-based rollback recovery enables a process to repeat its execution and be consistent with messages sent to outside world processes without having to take expensive checkpoints before sending such messages, additionally log-based recovery generally is not susceptible to the domino effect thereby allowing processes to use uncoordinated checkpointing if desired

state interval: pp.3 RHS, a process execution is a sequence of **state intervals** started by a nondeterministic event, execution during each state interval is deterministic, a concept related to the state interval is the piecewise deterministic assumption (PWD), this assumption states that

the system can detect and capture sufficient information about the nondeterministic events that initiate the state intervals
pp.6 RHS, a state interval is *recoverable* if there is sufficient information to replay the execution up to that state interval, a state interval is *stable* if the determinant of the nondeterministic event that started it is logged on stable storage, a recoverable state interval is always stable, but the opposite is not always true

consistent system state: pp.4 LHS, a **consistent system state** is one in which if a process's state reflects a message receipt, then the state of the corresponding sender reflects sending that message, inconsistent states occur because of failures (e.g. a process failed and restarted after sending a message), a fundamental goal of any rollback-recovery protocol is to bring the system into a consistent state when inconsistencies occur because of a failure, it is sufficient that the reconstructed state be one that could have occurred before the failure in a failure-free correct execution provided that it be consistent with the interactions that the system had with the outside world
pp.6 RHS, the most recent recoverable consistent system state is called the *maximum recoverable state*

outside world process: pp.4 RHS, to simplify the presentation of how rollback-recovery protocols interact with the outside world we model the latter as a special process that interacts with the rest of the system through message passing, this special process cannot fail, and it cannot maintain state or participate in the recovery protocol, furthermore since this special process models irreversible effects in the outside world, it cannot roll back, we call this special process the "outside world process" (OWP), before sending a message (output) to OWP the system must ensure that the state from which the message is sent will be recovered despite any future failure, this is commonly called the *output commit* problem, similarly input messages that a system receives from the outside world may not be reproducible during recovery because it may not be possible for OWP to regenerate them, thus recovery protocols must arrange to save these input messages so that they can be retrieved when needed for execution replay after a failure
rollback-recovery protocols therefore must provide special treatment for interactions with the outside world, there are two metrics that express the impact of this special treatment, namely the latency of input/output and the resulting slowdown of system's execution during input/output due to the overhead that the system incurs to ensure that its state will remain consistent with the messages exchanged with the OWP

in-transit message: pp.5 RHS, a message that has been sent but not yet received is called an *in-transit* message, if a system model assumes unreliable communication channels then the recovery protocol need not handle in-transit messages in any special way, since in-transit messages lost because of process failure cannot be distinguished from those lost because of communication failures in an unreliable communication channel

garbage collection: pp.7 LHS, garbage collection is the deletion of such useless recovery information, a common approach to garbage collection is to identify the most recent consistent set of checkpoints which is called the *recovery line*, and discard all information relating to events that occurred before that line

7.4 Checkpoint-Based Rollback Recovery

7.4.1 Uncoordinated checkpointing

pp.7 RHS, the processes record the dependencies among their checkpoints during failure-free operation, if a failure occurs the recovering process initiates rollback by broadcasting a *dependency request* message

to collect all the dependency information maintained by each process, when a process receives this message it stops its execution and replies with the dependency information saved on stable storage as well as with the dependency information if any associated with its current state, the initiator then calculates the recovery line based on the global dependency information and broadcasts a *rollback request* message containing the recovery line, upon receiving this message a process whose current state belongs to the recovery line simply resumes execution, otherwise it rolls back to an earlier checkpoint as indicated by the recovery line; pp.8 RHS, it can be shown under independent checkpointing that the maximum number of useful checkpoints that must be kept on stable storage cannot exceed $N(N+1)/2$ where N is the number of processes

7.4.2 Coordinated checkpointing

pp.9 RHS, coordinated checkpointing simplifies recovery and is not susceptible to the domino effect, also coordinated checkpointing requires each process to maintain only one permanent checkpoint on stable storage reducing storage overhead and eliminating the need for garbage collection, its main disadvantage however is the large latency involved in committing output, since a global checkpoint is needed before messages can be sent to OWP
pp.10 RHS, an example of a non-blocking checkpoint coordination protocol is the distributed snapshot in which markers play the role of the checkpoint-request messages, in this protocol the initiator takes a checkpoint and broadcasts a marker (a checkpoint request) to all processes, each process takes a checkpoint upon receiving the first marker and re-broadcasts the marker to all processes before sending any application message
pp.11 RHS, it is desirable to reduce the number of processes involved in a coordinated checkpointing session, this can be done since the processes that need to take new checkpoints are only those that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint, the following two-phase protocol achieves minimal checkpoint coordination

phase I: the checkpoint initiator identifies all processes with which it has communicated since the last checkpoint and sends them a request, upon receiving the request each process in turn identifies in processes it has communicated with since the last checkpoints and sends them a request and so on until no more processes can be identified

phase II: all processes identified in phase I take a checkpoint

in this protocol after a process takes a checkpoint it cannot send any message until phase II terminates successfully, although receiving a message after the checkpoint has been taken is allowed

7.4.3 Communication-induced checkpointing

pp.27 RHS, between the two ends of the coordinated and uncoordinated checkpointing are communication-induced checkpointing schemes that depend on the communication patterns of the applications to trigger checkpoints; pp.11 RHS, in communication-induced checkpointing (CIC) protocols processes take two kinds of checkpoints—local and forced, local checkpoints can be taken independently while forced checkpoint must be taken to guarantee the eventual progress of the recovery line, in particular CIC protocols take forced checkpoint to prevent the creation of useless checkpoints i.e. checkpoints that will never be part of a consistent global state, this decision can be formalized in an elegant theory based on the notions of Z-path and Z-cycle—a Z-path is a special sequence of messages that connects two checkpoints whereas a Z-cycle is a Z-path that begins and ends with the same checkpoint, Z-cycles are interesting in the context of CIC protocols because it can be proved that a checkpoint is useless if and only if it is part of a Z-cycle, hence one way to avoid useless checkpoints is to make sure that no Z-path ever becomes a Z-cycle
pp.12 RHS, traditionally CIC protocols have been classified in one of two types

model-based checkpointing: relies on preventing patterns of communications and checkpoints that could result in Z-cycles and useless checkpoints, a model is set up to detect the possibility that such patterns could be forming within the system according to some heuristic, it is possible that multiple processes detect the potential for inconsistent checkpoints and independently force local checkpoints to prevent the formation of undesirable patterns that may never actually materialize or that could be prevented by a single forced checkpoint, thus model-based checkpointing always errs on the conservative side by taking more forced checkpoints than is probably necessary

index-based coordination: assign timestamps to local and forced checkpoints such that checkpoints with the same timestamp at all processes form a consistent state; pp.13 LHS, it guarantees through forced checkpoints if necessary that (1) if there are two checkpoints $c_{i,m}$ and $c_{j,n}$ such that $c_{i,m} \mapsto c_{j,n}$ then $ts(c_{j,n}) \geq ts(c_{i,m})$ where $ts(c)$ is the timestamp associated with checkpoint c and (2) consecutive local checkpoints of a process have increasing timestamps, if checkpoints' timestamps always increase along a Z-path as opposed to simply non-decreasing as in (1) then no Z-cycle can ever form

recently it has been proved that the two types are fundamentally equivalent
pp.13 LHS, compared to other styles of checkpointing CIC protocols allows considerable autonomy in deciding when to take checkpoints, thus processes can take local checkpoints when their state is small and saving it incurs a small overhead, CIC protocols may also in theory scale up well in systems with a large number of processes since they do not require processes to participate in a globally coordinated checkpoint

7.5 Log-Based Rollback Recovery

pp.13 RHS, log-based rollback recovery makes explicit use of the fact that a process execution can be modeled as a sequence of deterministic state intervals each starting with the execution of a nondeterministic event, such an event can be the receipt of a message from another process or an event internal to the process, however sending a message is not a nondeterministic event; pp.14 LHS, a process p becomes an *orphan* when p itself does not fail, and p's state depend on the execution of a nondeterministic event e whose determinant cannot be recovered from stable storage or from the volatile memory of a surviving process; pp.13 RHS, log-based rollback recovery protocols guarantee that upon recovery of all failed processes the system does not contain any orphan process, i.e. a process whose state depends on a nondeterministic event that cannot be reproduced during recovery; pp.14 LHS, we call this property the *always-no-orphans* condition, it stipulates that if any surviving process depend on an event e, then either the event is logged on stable storage or the process has a copy of he determinant of event e, if neither condition is true then the process is an orphan because it depends on an event e that cannot be generated during recovery since its determinant has been lost

7.5.1 Pessimistic logging

pp.14 RHS, pessimistic logging protocols are designed under the assumption that a failure can occur after any nondeterministic event in the computation, in their most straightforward form pessimistic protocols log to stable storage the determinant of each nondeterministic event before the event is allowed to affect the computation, in a pessimistic logging system the observable state of each process is always recoverable, this property has four advantages

1. processes can send messages to the outside world without running a special protocol

2. processes restart from their most recent checkpoint upon a failure therefore limiting the extent of execution that has to be replayed, thus the frequency of checkpoints can be determined by trading off the desired runtime performance which the desired protection of the on-going execution

3. recovery is simplified because the effects of a failure are confined only to the processes that fail, functioning processes continue to operate and never become orphans because a process always recovers to the state that included its most recent interaction with any other process including OWP

4. garbage collection is simple, older checkpoints and determinants of nondeterministic events that occurred before the most recent checkpoint can be reclaimed

pp.15 LHS, the price to be paid for these advantages is a performance penalty incurred by synchronous logging, this overhead can be lowered using special hardware; pp.15 RHS, the performance overhead of pessimistic logging can also be reduced by delivering a message or an event and deferring its logging until the receiver communicates with any other process including OWP, this property relaxes the condition of pessimistic logging by allowing a single process to be affected by an event that has yet to be logged, provided that the process does not externalize the effect of this dependency to other processes including OWP, event logging and delivery are not performed in one atomic operation in this variation of pessimistic logging, this scheme reduces overhead because several events can be logged in one operation reducing the frequency of synchronous access to stable storage

7.5.2 Optimistic logging

pp.16 LHS, in optimistic logging protocols process log determinants asynchronously to stable storage, these protocols make the optimistic assumption that logging will complete before a failure occurs, determinants are kept in a volatile log which is periodically flushed to stable storage, thus optimistic logging does not require the application to block waiting for the determinants to be actually written to stable storage and therefore incurs little overhead during failure-free execution; pp.16 RHS, however if a process fails the determinants in its volatile log will be lost and the state intervals that were started by the nondeterministic events corresponding to these determinants cannot be recovered, furthermore if the failed process sent a message during any of the state intervals that cannot be recovered, the receiver of the message becomes an orphan process and must roll back to undo the effects of receiving the message
pp.16 RHS, optimistic protocols do not implement the always-on-orphans condition and therefore permit the temporary creation of orphan processes, however they require that the property holds by the time recovery is complete, this is achieved during recovery by rolling back orphan processes until their states do not depend on any message whose determinant has been lost, to perform these rollbacks correctly optimistic logging protocols track causal dependencies during failure-free execution, upon a failure the dependency information is used to calculate and recover the latest global state of the pre-failure execution in which no process is in an orphan, hence while pessimistic protocols need only keep the most recent checkpoint of each process, optimistic protocols may need to keep multiple checkpoints, moreover since determinants are logged asynchronously output commit in optimistic logging protocols generally requires multi-host coordination to ensure that no failure scenario can revoke the output
pp.17 LHS, recovery in optimistic logging protocols can be either synchronous or asynchronous

synchronous: pp.17 LHS, all processes run a recovery protocol to compute the maximum recoverable system state based on dependency and logged information and then perform the actual rollbacks, dependency tracking can be either direct or transitive

- **direct:** the state interval index of the sender is piggybacked on each outgoing message to allow the receiver to record the dependency directly caused by the message
- **transitive:** each process P_i maintains a size-N vector TD_i where $TD_i[i]$ is P_i's current state interval index and $TD_i[j]$, $j \neq i$, records the highest index of any state interval of P_j on which P_i depends

transitive dependency tracking generally incurs a higher failure-free overhead for piggybacking and maintaining the dependency vectors but allows faster output commit and recovery

asynchronous: pp.17 RHS, a failed process restarts by sending a rollback announcement broadcast or a recovery message broadcast to start a new *incarnation*, upon receiving a rollback announcement a process rolls back if it detects that it has become an orphan with respect to that announcement and then broadcast its own rollback announcement, since rollback announcements from multiple incarnations of the same process may coexist in the system each process in general needs to track the dependency of its state on every incarnation of all processes to correctly detect orphaned states, a way to limit dependency tracking to only one incarnation of each process is to force a process to delay its delivery of certain messages, piggybacking all rollback announcements known to a process on every outgoing message can eliminate blocking
another issue in asynchronous recovery protocols is the possibility of *exponential rollbacks*, this phenomenon occurs if a single failure causes a process to roll back an exponential number of times; pp.18 LHS, exponential rollbacks can be eliminated by distinguishing failure announcements from rollback announcements and by broadcasting only the former, another possibility is to piggyback the original rollback announcement from the failed process on every subsequent rollback announcement that it triggers

7.5.3 Causal logging

pp.18 LHS, causal logging has the failure-free performance advantages of optimistic logging while retaining most of the advantages of pessimistic logging

- like optimistic logging it avoids synchronous access to stable storage except during output commit
- like pessimistic logging it allows each process to commit output independently and ever creates orphans thereby isolating each process from the effects of failures that occur in other processes, furthermore causal logging limits the rollback of any failed process to the most recent checkpoint on stable storage

pp.18 RHS, causal logging protocols ensure the always-no-orphans property by ensuring that the determinant of each nondeterministic event that causally precedes the state of a process is either logged on stable storage or is available in the volatile log of the process; pp.19 LHS, thus a process does not need to run a multi-host protocol to commit output, causal logging protocols implements the always-no-orphans condition by having processes piggyback the non-stable determinants in their volatile log on the messages they send to other processes
pp.19, table 1 below summarizes a comparison between the different variations of rollback recovery protocols

7.6 Implementation

pp.20 RHS, the few experimental studies available have shown that building rollback recovery protocols with low failure-free overhead is feasible, these studies also provide ample evidence that the main difficulty in implementing these protocols lies in the complexity of handling recovery, it is interesting to note that all commercial implementations of message logging use pessimistic logging because it simplifies recovery
pp.21 LHS, recent results have shown that

- stable storage access is now the major source of overhead in checkpointing or message logging systems, communication overhead is much lower in comparison, such changes favor coordinated checkpointing schemes over message logging or uncoordinated checkpointing systems as they require less access to stable storage and are simpler to implement

	Uncoordinated Checkpointing	Coordinated Checkpointing	Comm. Induced Checkpointing	Pessimistic Logging	Optimistic Logging	Causal Logging
PWD assumed?	No	No	No	Yes	Yes	Yes
Checkpoint/process	Several	1	Several	1	Several	1
Domino effect	Possible	No	No	No	No	No
Orphan processes	Possible	No	Possible	No	Possible	No
Rollback extent	Unbounded	Last global checkpoint	Possibly several checkpoints	Last checkpoint	Possibly several checkpoints	Last checkpoint
Recovery data	Distributed	Distributed	Distributed	Distributed or local	Distributed or local	Distributed
Recovery protocol	Distributed	Distributed	Distributed	Local	Distributed	Distributed
Output commit	Not possible	Global coordination required	Global coordination required	Local decision	Global coordination required	Local decision

Table 1 A comparison between various flavors of rollback-recovery protocols.

Figure 1: a comparison between various flavors of rollback recovery protocols

- arbitrary forms of nondeterminism can be supported at a very low overhead in logging systems

7.6.1 Checkpointing implementation

pp.21 LHS, all available studies have shown that writing the state of a process to stable storage is the largest contributor to the performance overhead, several techniques exist to reduce this overhead

concurrent checkpointing: relies on the memory protection hardware available in modern computer architectures to continue the execution of the process while its checkpoint is being saved on stable storage

incremental checkpointing: avoids rewriting portions of the process states that do not change between consecutive checkpoints, it can be implemented by using the dirty-bit of the memory protection hardware or by emulating a dirty-bit in software, incremental checkpointing can also be extended over several processes, in this technique the system saves the computed parity or some function of the memory pages that are modified across several processes, this technique is very similar to parity computation in RAID disk systems, the parity pages can be saved in volatile memory of some other processes thereby avoiding the need to access stable storage

pp.22 LHS, support for checkpointing can be implemented in the kernel or it can be implemented by a library linked with the user program

kernel level: more powerful because they can also capture kernel data structures that support the user process, however these implementations are necessarily not portable

user level: system calls that manipulate memory protection such as mprotect of UNIX can emulate concurrent and incremental checkpointing, the fork system call of UNIX can implement concurrent checkpointing if the operating system implements fork using copy-on-write protection

pp.22 LHS, a compiler can be instrumented to generate code that supports checkpointing, the compiled program contains code that decides when and what to checkpoint, the advantage of this technique is that the compiler can decide on the variables that must be saved therefore avoiding unnecessary data, furthermore the compiler may decide the points during program execution where the amount of state to be saved is small, compiler support could also be simplified in languages that support automatic garbage collection, as the execution point after each major garbage collection provides a convenient place to take a checkpoint at a minimum cost
pp.22 RHS, a large amount of work has been devoted to analyzing and deriving the optimal checkpointing frequency and placement, the problem is usually formulated as an optimization problem subject to constraints, however generally it has been observed in practice that the overhead of checkpointing is usually negligible, therefore the optimality of checkpoint placement is rarely an issue in pratical systems

7.6.2 Checkpointing protocols in comparison

pp.22 RHS, using concurrent and incremental checkpointing the overhead of either coordinated or uncoordinated checkpointing is essentially the same, it follows that coordinated checkpointing is superior to uncoordinated checkpointing when all aspects are considered on the balance
pp.22 RHS, communication-induced checkpointing does not scale well as the number of processes increases, the occurrence of forced checkpoints at random points within the execution due to communication messages makes it very difficult to predict the required amount of stable storage for a particular application run, also this unpredictability affects the policy for placing local checkpoints and makes communication-induced protocols cumbersome to use in practice

7.6.3 Communication protocols

pp.23 LHS, rollback recovery complicates the implementation of protocols used for inter-process communications, for all communication protocols a rollback recovery system must mask the actual identity and location of processes or remote ports from the application program, this masking is necessary to prevent any application program from acquiring a dependency on the location of a certain process, making it impossible to restart the process on a different machine after a failure, a solution to this problem is to assign a logical, location-independent identifier to each process in the system
pp.23 LHS, after a failure identity masking and communication redirection are sufficient for communication protocols that offer the abstraction of an unreliable channel, protocols that offer the abstraction of reliable channels require additional support, these protocols usually generate a timeout upcall to the application program if the process at the other end of the channel has failed, these timeouts should be masked, masking timeouts should also be coupled with the ability of the protocol implementation to reestablish the connection with the restarting process; pp.23 RHS, recovering in-transit messages that are lost because of a failure is another problem for reliable communication protocols, messages must be saved at the sender side for possible retransmission during recovery

7.6.4 Log-based recovery

pp.23 RHS, message logging introduces three sources of overhead

1. each message must in general be copied to the local memory of the process
2. the volatile log is regularly flushed to stable storage to free up space

3. message logging nearly doubles the communication bandwidth required to run the application for systems that implement stable storage via a highly available file system accessible through the network

pp.23 RHS, the first source of overhead may directly affect communication throughput and latency, this is especially true if the copying occurs in the critical path of the inter-process communication protocol, in this respect sender-based logging is considered more efficient than receiver-based logging because (1) the copying can take place after sending the message over the network (2) the system may combine the logging of messages with the implementation of the communication protocol and share the message log with the transmission buffers which avoids the extra copying of the message (3) another optimization for sender-based logging systems is to use copy-on-write which allows the system to have one copy for identical messages
pp.24 LHS, the combination of event logging with coordinated checkpointing obviates the need for flushing the volatile message logs to stable storage in a sender-based logging implementation

7.6.5 Stable storage

pp.24 LHS, magnetic disks have been the medium of choice for implementing stable storage, although they are slow their storage capacity and low cost combination cannot be matched by other alternatives
pp.24 LHS, an implementation of a stable storage abstraction on top of a conventional file system may not be the best choice, because modern file systems tend to be optimized for the pattern of access expected in workstation or personal computing environments, furthermore these file systems are often accessed through a network via a protocol that is optimized for small file accesses and not for large file accesses that are more common in checkpointing and logging, thus an implementation of stable storage should bypass the file system layer and access the disk directly

7.6.6 Support for nondeterminism

pp.24 RHS, nondeterminism occurs when the application program interacts with the operating system through system calls and upcalls (asynchronous events), system calls in general can be classified into three types

- those that do not need to be logged, e.g. idempotent system calls that return deterministic values whenever executed including calls that return the user identifier of the process owner
- those that must be logged during failure-free operation but should not be re-executed during execution replay, because re-executing these calls during recovery might return a different value that is inconsistent with the pre-failure execution, e.g. calls that inquire about the environment such as getting the current time of day, the result from these calls should simply be replayed to the application program
- those that must be logged during failure-free operation and re-executed during execution replay, because these calls generally modify the environment and therefore they must be re-executed to re-establish the environment changes, e.g. calls that allocate memory or create processes

pp.24 RHS, nondeterminism results from asynchronous signals available in the form of software interrupts under various operating systems, such signals must be applied at the same execution points during replay to reproduce the same result, log-based rollback recovery can cover this form of nondeterminism by taking a checkpoint after the occurrence of each signal but this can be very expensive, alternatively the system may convert these asynchronous signals to synchronous messages, or it may queue the signals until the application polls for them, both alternatives convert asynchronous event notifications into synchronous ones, which may not be suitable or efficient for many applications

pp.25 LHS, another example of nondeterminism that is difficult to track is shared memory manipulation in multi-threaded applications, reconstructing the same execution during replay requires the same interleaving of shared memory accesses by the various threads as in the pre-failure execution, systems that support this form of nondeterminism supply their own sets of locking primitives and require applications to use them for protecting access to shared memory, however it makes shared memory access expensive and may generate a large volume of data in the log
pp.25 LHS, a technique for tracking nondeterminism due to asynchronous signals and interleaved memory access on single processor system is to use *instruction counters*, an instruction counter is a register that decrements by one upon the execution of each instruction leading the hardware to generate an exception when the register contents become 0, an instruction counter can be used in two modes, in one mode the register is loaded with the number of instructions to be executed before a breakpoint occurs, in the second mode execution proceeds until an event of interest occurs at which time the content of the counter is sampled and the number of instructions executed since the time the counter was set is computed and logged, instruction counters can be used in rollback recovery to track the number of instructions that occur between asynchronous interrupts, an instruction counter can be implemented in hardware, it also can be emulated in software

7.6.7 Dependency tracking

pp.25 RHS, when dependency tracking requires more complex structures there are techniques for reducing the amount of actual data that need to be transferred on top of each message, all these techniques revolve around two themes, first only incremental changes need to be sent, implementation of this optimization is straightforward in systems that assume FIFO communication channels, when lossy channels are assumed this optimization is still possible but at the expense of more processing overhead, the other technique for reducing the overhead of dependency tracking exploits the semantics of the applications and the communication patterns, for instance if if can be inferred from the dependency information available to process p that process q already knows parts of the information that is to be piggybacked on a message outgoing to q, then process q can exclude this information, regardless of the particular method used to track inter-process dependencies various prototype implementations have shown that the overhead resulting from dependency tracking is negligible compared to the overhead of checkpointing or logging

7.6.8 Recovery

pp.26 LHS, the simplest solution to implanting a process in a different environment during recovery is to attempt to restart the program on the same host, if this is not feasible then the system must insulate the process from environment-specific variables, another problem in implementing recovery is the need to reconstruct the auxiliary state within the operating system kernel that supports the recovering process, for protocols implemented outside the operating system the rollback recovery system must emulate these data structures and log sufficient information to be able to recreate them during recovery; pp.26 RHS, since most of the applications that benefit from rollback recovery seem to be in the realm of scientific computing where no sophisticated state is maintained by the kernel on behalf of the processes, this problem does not seem to be severe in that particular context
pp.26 RHS, it has been observed that for log-based recovery systems the messages and determinants available in the logs are replayed at a considerably higher speed during recovery than during normal execution because a process needs not block waiting for messages or synchronization events, it also has been observed that sender-based logging protocols typically slow down recovery if there are multiple failures because of the need to re-execute some of the processes under control to regenerate the messages

7.6.9 Checkpointing and mobility

pp.26 RHS, in mobile computing the fundamental concepts of distributed checkpointing, consistency, and rollback are identical to those in traditional distributed systems, however energy constraints, intermittent communications, and low-performance processors of mobile computing favor checkpointing protocols that allow maximum autonomy to participating processes, require low overhead in resources, and can function with the minimum possible number of message exchanges, therefore independent checkpointing and communication-induced checkpointing tend to be more appropriate for these environments than coordinated checkpointing, also log-based recovery protocols that allow a high degree of autonomy during recovery tend to be more appropriate for these environments than those protocols that require global communication during recovery, nevertheless checkpointing and rollback recovery have yet to prove useful for mobile hosts

7.6.10 Rollback recovery in practice

pp.27 LHS, despite the wealth of research in the area of rollback recovery in distributed systems very few commercial systems actually have adopted them, difficulties in implementing recovery perhaps are the main reason why these protocols have not been widely adopted, additionally the range of applications that benefit from these protocols tend to be in the realm of long-running scientific programs which are relatively few, log-based recovery seemed to have less success than checkpoint-only systems

8 Lesson #8 Paxos Made Simple

8.1 The Publication Details

Author: Leslie Lamport

Title: Paxos Made Simple

Journal: ACM SIGACT News December 2001, pp.51–58

DOI: `https://www.microsoft.com/en-us/research/publication/2016/12/paxos-simple-Copy.pdf`

8.2 Derivation

safety requirement: pp.1, the safety requirements for consensus are

- only a value that has been proposed may be chosen
- only a single value is chosen
- a process never learns that a value has been chosen unless it actually has been

the goal is to ensure that some proposed value is eventually chosen, and if a value has been chosen then a process can eventually learn the value

three roles in the consensus algorithm: pp.1, we let the three roles in the consensus algorithm be performed by three classes of agents: *proposers*, *acceptors*, and *learners*, in an implementation a single process may act as more than one agent

prepare request: pp.4, a proposer choose a new proposal number n and sends a request to each member of some set of acceptors, asking it to respond with

- a promise never again to accept a proposal numbered less than n
- the proposal with the highest number less than n that it has accepted if any

choose value: pp.4, if the proposer receives the requested responses from a majority of the acceptors, then it can issue a proposal with number n and value v, where v is the value of the highest-numbered proposal among the responses, or is any value selected by the proposer if the responders reported no proposals

accept request: pp.5, a proposer issues a proposal by sending a request that the proposal be accepted to some set of acceptors, which need not be the same set of acceptors that responded to the initial request, an acceptor can accept a proposal numbered n if and only if it has not responded to a prepare request with a number greater than n

acceptor optimization: pp.5, suppose an acceptor receives a prepare request numbered n but it has already responded to a prepare request numbered greater than n, the acceptor can ignore such a prepare request, it can also ignore a prepare request for a proposal it has already accepted, hence an acceptor needs to remember only the highest-numbered proposal that it has ever accepted and the number of the highest-numbered prepare request to which it has responded

proposer optimization: pp.6, it is probably a good idea to abandon a proposal if some proposer has begun trying to issue a higher-numbered one, therefore if an acceptor ignores a prepare or accept request because it has already received a prepare request with a higher number, then it should probably inform the proposer who should then abandon its proposal

8.3 Choosing a Value

pp.5, the algorithm operates in the following two phases

phase I—prepare request: a proposer selects a proposal number n and sends a prepare request with number n to a majority of acceptors
if an acceptor receives a prepare request with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal that it has accepted if any

phase II—accept request: if the proposer receives a response to its prepare requests numbered n from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses or any value if the responses reported no proposals
if an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n

8.4 Learning a Chosen Value

pp.6, the assumption of non-Byzantine failures makes it easy for one learner to find out from another learner that a value has been accepted, we can have the acceptors respond with their acceptances to a distinguished learner, which in turn informs the other learners when a value has been chosen, this approach requires an extra round for all the learners to discover the chosen value, it is also less reliable since the distinguished learner could fail, but it requires a number of responses equal only to the sum of the number of acceptors and the number of learners, more generally the acceptors could respond with their acceptances to some set of distinguished learners each of which can then inform all the learners

when a value has been chosen, using a larger set of distinguished learners provides greater reliability at the cost of greater communication complexity
pp.7, because of message loss a value could be chosen with no learner ever finding out, the learner could ask the acceptors what proposals they have accepted, but failure of an acceptor could make it impossible to know whether or nor a majority had accepted a particular proposal, in that case learners will find out what value is chosen only when a new proposal is chosen using the algorithm described above

8.5 Progress

pp.7, it's easy to construct a scenario in which two proposers each keep issuing a sequence of proposals with increasing numbers, none of which are ever chosen, to guarantee progress a distinguished proposer must be selected as the only one to try issuing proposals, by abandoning a proposal and trying again if it learns about some request with a higher proposal number, the distinguished proposer will eventually choose a high enough proposal number, if enough of the system (proposer, acceptors, and communication network) is working properly, liveness can therefore be achieved by electing a single distinguished proposer, a reliable algorithm for electing a proposer must use either randomness or real time (e.g. by using timeouts), however safety is ensured regardless of the success or failure of the election

8.6 Implementation

pp.7, the algorithm chooses a leader which plays the roles of the distinguished proposer and the distinguished learner, an acceptor records its intended response in stable storage before actually sending the response, all that remains is to describe the mechanism for guaranteeing that no two proposals are ever issued with the same number, this can be achieved by

- different proposers choose their numbers from disjoint sets of numbers, so two different proposers never issue a proposal with the same number
- each proposer remembers in stable storage the highest-numbered proposal it has tried to issue, and begins phase I with a higher proposal number than any it has already used

8.7 Paxos Consensus Algorithm as a State Machine

pp.8, a simple way to implement a distributed system is as a collection of clients that issue commands to a central server, an implementation that uses a single central server fails if that server fails, we therefore instead use a collection of servers each one independently implementing the state machine, because the state machine is deterministic, all the servers will produce the same sequences of states and outputs if they all execute the same sequence of commands, a client issuing a command can then use the output generated for it by any server, to guarantee that all servers execute the same sequence of state machine commands, we implement a sequence of separate instances of the Paxos consensus algorithm, the value chosen by the ith instance being the ith state machine command in the sequence, each server plays all the roles (proposer, acceptor, and learner) in each instance of the algorithm
pp.9, in normal operation a single server is elected to be the leader which acts as the distinguished proposer (the only one that tries to issue proposals) in all instances of the consensus algorithm, who decides where in the sequence each client command should appear, it might fail because of failures or because another server also believes itself to be the leader and has a different idea of what the kth command should be, but the consensus algorithm ensures that at most one command can be chosen as the kth one
pp.9, when the previous leader has just failed and a new learner has been selected, suppose it knows commands 1–134, 138, and 139

1. it executes phase I of instances 135–137 and of all instances greater than 139

2. suppose that the outcome of step 1 above determine the value to be proposed in instances 135 and 140 but leaves the proposed value unconstrained in all other instances, it then executes phase II for instances 135 and 140 thereby choosing commands 135 and 140

3. it can now execute commands 1–135, however it can't execute commands 138–140 because commands 136 and 137 have yet to be chosen, we let it fill the gap immediately by proposing as commands 136 and 137 a special "no-op" command that leaves the state unchanged by executing phase II of instances 136 and 137, once no-op commands have been chosen commands 138–140 can be executed

4. it can propose command 142 before it learns that its proposed command 141 has been chosen, when it fails to receive the expected response to its phase II messages in instance 141, it will retransmit those messages, it could fail again leaving a gap (command 141) in the sequence in chosen command

pp.10, because (1) a leader can executes phase I by sending a single reasonably short message to the other servers, in phase I an acceptor responds with more than a simple OK only if it has already received a phase II message from some proposer (2) failure of the leader and election of a new one should be rare events, the effective cost of executing a state machine command is the cost of executing only phase II of the consensus algorithm, and it can be shown that phase II of the Paxos consensus algorithm has the minimum possible cost of any algorithm for reaching agreement in the presence of faults, hence the Paxos algorithm is essentially optimal
pp.10, in abnormal circumstances the leader election might fail, if no server is acting as leader then no new commands will be proposed, if multiple servers think they are leaders then they can all propose values in the same instance of the consensus algorithm, which could prevent any value from being chosen, however safety is preserved—two different servers will never disagree on the value chosen as the ith state machine command, election of a single leader is needed only to ensure progress
pp.11, if the set of servers can change, the current set of servers can be made part of the state and can be changed with ordinary state-machine commands

8.8 Key Facts & Ideas

asynchronous non-Byzantine model: pp.2, we use the customary asynchronous non-Byzantine model in which

- agents operate at arbitrary speed, may fail by stopping, and may restart, since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted
- messages can take arbitrarily long to be delivered and can be lost, but they are not corrupted

9 Lesson #8 A Brief Analysis of Consensus Protocol: From Logical Clock to Raft

9.1 The Publication Details

Author: Alibaba Cloud

Title: A Brief Analysis of Consensus Protocol: From Logical Clock to Raft

Journal: N.A.

DOI: https://www.alibabacloud.com/blog/a-brief-analysis-of-consensus-protocol-from-logical-594675

9.2 Logical Clock

pp.1, a logical clock is used to define the order of occurrence for two associated events, a logical clock cannot determine the order of occurrence for unassociated events, so this "happens before" relation is actually a partial ordering relation
pp.2, the algorithm of the logical clock is as follows (each event corresponds to a Lamport timestamp, the initial value is 0):

- if an event occurs within a node, the timestamp adds 1
- if an event is a sending event, the timestamp adds 1 and the timestamp is added to that message
- if an event is a reception event, the timestamp is max(local timestamp within a message) + 1

9.3 Replicated State Machine

pp.3, usually state machine replication and consensus protocol algorithms are used together to implement high availability and fault tolerance in distributed systems, state machine replication maintains a persisted log in each instance copy in a distributed system and uses a certain consensus protocol algorithm to ensure that the log is completely consistent within each instance

9.4 Paxos

pp.4, it is impossible to directly use Paxos for state machine replication, instead we need to add many things to Paxos, that is why Paxos has so many variants

9.5 Basic Paxos

pp.4, Paxos can only enable consensus for one value and the proposal cannot be changed once decided

prepare phase: a proposer sends a propose ID n to each acceptor

- if an acceptor finds that it has never received a value greater than n from any of the proposers, it will reply to the proposer and promise not to receive proposals with propose ID smaller than or equal to n
- if an acceptor has already promised a proposed number greater than n, it will not reply to the proposer
- if an acceptor has accepted a proposal less than n, it will return this proposed value to the proposer, otherwise a null value is returned

when the proposer receives replies from more than half of the acceptors, it is ready to start the accept phase

accept phase: the proposer can propose a new value if and only if the replies received do not include values of previous proposals, otherwise the proposer can only use the greatest proposal value among the replies as the proposal value, the proposer uses this value and propose ID n to launch the accept request against each acceptor, however the accept may fail because the acceptor may have given a promise to a proposer with a higher propose ID before the accept request is initiated, in other words the second phase may still fail due to the presence of multiple proposers

9.6 Multi-Paxos

pp.6, Paxos cannot be directly used for state machine replication, the reasons are as follows

1. Paxos can only determine one value and cannot be used for continuous log replication
2. the presence of multiple proposers may lead to livelock, multiple submittals of proposals may be required
3. the final result of a proposal is only known to partial acceptors, this cannot guarantee that each instance for state machine replication has a completely consistent log

pp.6, multi-Paxos is used to solve the three aforementioned problems

1. each index value of a log entry uses an independent Paxos instance
2. include only one proposer in a Paxos group, select a leader
3. add a firstUnchosenIndex to each server and let a leader synchronize selected values to each acceptor

9.7 ZAB

pp.6, ZAB (Zookeeper Atomic Broadcast) is a consensus protocol used in Zookeeper, ZAB is a dedicated protocol for Zookeeper; pp.7, Zookeeper did not directly use Paxos but developed its own protocol because Paxos was considered incapable of meeting the requirements of Zookeeper, e.g. Paxos allowing multiple proposers may cause multiple commands submitted from a client to fail to be executed by FIFO sequence, additionally in the recovery process the data of some followers may be incomplete, in fact these problems have been solved in some variants of Paxos
pp.7, ZAB enables log replication by using a two-phase commit

vote phase: successfully complete when more than half of the consent votes are obtained to ensure that more than half of the machines are working correctly or within the same network partition

commit phase: data is transferred to each follower and then each follower (as well as leader) appends data to the log

if the leader fails in the commit phase and this write operation has been committed on at least on follower, this follower will definitely be selected as a leader because its zxid (epoch number and transition number) is the greatest, if no followers have committed this message when the leader fails this write operation is not completed
pp.8, ZAB supports stale reads from replicas, to implement strongly consistent reading we can use sync read—first a virtual write operation is launched (nothing is written), after this operation is completed, this sync operation is also committed locally, then reading is performed on the local replicas to ensure that all the data before this sync time point is correctly read

9.8 Raft

pp.8, the developers of Raft listed some disadvantages of Paxos

1. leader selection and proposal are mixed together in Paxos making Paxos hard to understand, while leader selection and consensus agreement are separated at the very beginning of designing Raft

2. the original Paxos protocol is only to reach consensus on one single event, once a value is determined it cannot be modified, however in realistic scenarios it is required to continuously reach consensus on the value of a log entry, although the Paxos protocol has been proven by Lamport, the Paxos-based and improved algorithms like multi-Paxos are unproven

3. Paxos only provides a rough description, this requires subsequent improvements on Paxos, to apply Paxos in your own projects basically you have to customize and implement a set of Paxos protocols that meet your specific requirements

pp.9, Raft uses a very simple leader selection solution: each server sleeps for a certain period of time and the server that wakes up first makes a proposal, if this server receives the majority of the votes it is selected as the leader, in a general network environment the server that makes a vote first will also receive votes from other votes first, therefore only one round of votes are required to select a leader
pp.9, a total of two RPC calls are required for interaction between the leader and followers if snapshots and changes in the number of members are not taken into consideration, one of the two calls is the RequestVote RPC which is only required for leader selection, all data interactions are performed by the AppendEntries RPC
pp.9, each leader has its own term which will be applied in each entry of the log to represent which term that entry is written in (done by the AppendEntries RPC call), a term is equivalent to a lease, if a leader does not send a heartbeat (also done by the AppendEntries RPC call) within a specified period of time, a follower will consider that the leader has failed and add 1 to the largest term that it has received to form a new term and start a new election, if no followers receive enough votes before the timeout, a follower will add 1 to the current term and start a new vote request until a new leader is selected
pp.10, the write operation of the leader is also a two-phase commit process, first the leader will write the first vacant index found in its own log and send the value of that entry to each follower through the AppendEntries RPC, if the leader receives "true" from more than half of the followers (including itself) the leader adds 1 to committed index in the next AppendEntries indicating that the written entry has been submitted, if a follower returns "false" to the leader the leader will traverse backwards indefinitely until it finds an entry that is consistent with the log content of the follower, from that point the log content of the leader is re-sent to the follower to complete the recovery
pp.11, the first interesting aspect is that log entries in Raft can be modified, if a leader fails the newly elected leader may reuse this index and the index content of this follower may be modified, this causes two problems: logs in Raft cannot be implemented in an append-only file or file system (vs. for ZAB and Paxos protocols logs are only appended), the second interesting aspect is that only one committed index is maintained in Raft to ensure simplicity, any entries that are smaller or equal to this committed index will be considered to have been committed

9.9 Closing Words

pp.13, it is generally believed that the Raft protocol has lower performance than Paxos because it only allows committing entries in sequence

10 Lesson #9 Spanner: Google's Globally-Distributed Database

10.1 The Publication Details

Author: James C. Corbett et al.

Title: Spanner: Google's Globally-Distributed Database

Journal: Proceedings of OSDI 2012, pp.251–264

DOI: https://static.googleusercontent.com/media/research.google.com/en//archive/spanner-osdi
pdf

10.2 Introduction

pp.1 LHS, Spanner is Google's scalable multi-version globally-distributed and synchronously-replicated database, at the highest level of abstraction it is a database that shards data across many sets of Paxos state machines; pp.1 RHS, Spanner's main focus is managing cross-datacenter replicated data, data is stored in schematized semi-relational tables, data is versioned, and each version is automatically timestamped with its commit time, old versions of data are subject to configurable garbage-collection policies and applications can read data at old timestamps, Spanner supports general-purpose transactions and provides a SQL-based query language
pp.1 RHS, Spanner provides several interesting features

- the replication configurations for data can be dynamically controlled at a fine grain by applications, e.g. which datacenter to store, how far data is from users (to control read latency), how far replicas are from each other (to control write latency), and how many replicas
- Spanner has two features that are difficult to implement in a distributed database—externally consistent reads and writes and globally-consistent reads across the database at a timestamp, which enable Spanner to support consistent MapReduce executions and atomic schema updates

pp.2 LHS, these features are enabled by the fact that Spanner assigns globally-meaningful commit timestamps to transactions which reflect serialization order, the key enabler of these properties is a new TrueTime API which keeps uncertainty small by using multiple modern clock references (GPS and atomic clocks)

10.3 Implementation

pp.2 LHS, a Spanner deployment is called a *universe*, each universe contains a set of *zones*; pp.2 RHS, zones are the unit of administrative deployment, the set of zones is also the set of locations across which data can be replicated, zones are also the unit of physical isolation—there may be one or more zones in a datacenter, a zone has one *zonemaster* and between one hundred and several thousand *spanservers*, the former assigns data to spanservers while the latter serve data to clients, each spanserver is responsible for between 100 and 1000 instances of a data structure called a *tablet*

Spanserver software stack: pp.3 LHS, to support replication each spanserver implements a single Paxos state machine on top of each tablet, our Paxos implementation supports long-lived leaders with time-based leader leases whose length defaults to 10 seconds, the current Spanner implementation logs every Paxos write twice—once in the tablet's log and once in the Paxos log, our implementation of Paxos is pipelined so as to improve Spanner's throughput in the presence of WAN latencies, but writes are applied by Paxos in order, writes must initiate the Paxos protocol at the leader, reads access state directly from the underlying tablet at any replica that is sufficiently up-to-date, the set of replicas is collectively a *Paxos group*
at every replica that is a leader, each spanserver implements a *lock table* to implement concurrency control, the lock table contains the state for two-phase locking—it maps ranges of keys to lock states, each spanserver also implements a *transaction manager* to support distributed transactions, if a transaction involves only one Paxos group it can bypass the transaction manager, if a transaction involves more than one Paxos group those groups' leaders coordinate to perform

two-phase commit; pp.4 RHS, running two-phase commit over Paxos mitigates the availability problems

directories and placement: pp.3 LHS, the Paxos state machines are used to implement a consistently replicated bag of mappings, the key-value mapping state of each replica is stored in its corresponding tablet; pp.3 RHS, on top of the bag of key-value mappings the Spanner implementation supports a bucketing abstraction called a *directory*, which is a set of contiguous keys that share a common prefix, supporting directories allows applications to control the locality of their data by choosing keys carefully, a directory is the unit of data placement, when data is moved between Paxos groups it is moved directory by directory; pp.4 LHS, a directory is also the smallest unit whose geographic replication properties can be specified by an application (administrators control two dimensions—the number and types of replicas and the geogrpahic placement of those replicas), the fact that a Paxos group may contain multiple directories implies that a Spanner tablet is a container that may encapsulate multiple partitions of the row space

data model: pp.4 LHS, Spanner exposes the following set of data features to applications: a data model based on schematized semi-relational tables, a query language, and general-purpose transactions, the need to support schematized semi-relational tables and synchronous replication is supported by the popularity of Megastore because of its support for synchrornous replication across datacenters (vs. Bigtable only supports eventually-consistent replication across datacenters); pp.4 RHS, Spanner's data model is not purely relational in that rows must have names, more precisely every table is required to have an ordered set of one or more primary-key columns, this requirement is where Spanner still look like a key-value store—the primary keys form the name for a row and each table defines a mapping from the primary-key columns to the non-primary-key columns; pp.5 LHS, the interleaving of tables to form directories is significant because it allows clients to describe the locality relationships that exist between multiple tables

10.4 TrueTime

pp.5 LHS, TrueTime explicitly represents time as a *TTinterval* which is an interval with bounded time uncertainty (unlike standard time interfaces that give clients no notion of uncertainty); pp.5 RHS, the underlying time references used by TrueTime are GPS and atomic clocks, TrueTime uses two forms of time reference because they have different failure modes, TrueTime is implemented by a set of *time master* machines per datacenter and a *timeslave daemon* per machine, the majority of masters have GPS receivers with dedicated antennas, the remaining masters are equipped with atomic clocks, every daemon polls a variety of masters to reduce vulnerability to errors from any one master

10.5 Concurrency Control

10.5.1 Timestamp management

pp.6 LHS, the Spanner implementation supports (1) read-write transactions (2) read-only transactions (3) snapshot reads, a read-only transaction must be predeclared as not having any writes thus it is not simply a read-write transaction without any writes, reads in a read-only transaction execute at a system-chosen timestamp without locking so that incoming writes are not blocked, a snapshot read is a read in the past that executes without locking

Paxos leader leases: pp.6 RHS, Spanner's Paxos implementation uses timed leases to make leadership long-lived (10 seconds by default), a potential leader sends requests for timed *lease votes*, upon receiving a quorum of lease votes the leader knows it has a lease, define a leader's *lease interval* as starting when it discovers it has a quorum of lease votes, and as ending when it no

longer has a quorum of lease votes (because some have expired), Spanner depends on the following **disjointness invariant**: for each Paxos group each Paxos leader's lease interval is disjoint from every other leader's

assigning timestamps to RW transactions: pp.6 RHS, transactional reads and writes use two-phase locking, as a result they can be assigned timestamps at any time when all locks have been acquired but before any lock have been released, for a given transaction Spanner assigns it the timestamp that Paxos assigns to the Paxos write that represents the transaction commit, Spanner depends on the following **monotonicity invariant**: within each Paxos group Spanner assigns timestamps to Paxos writes in monotonically increasing order even across leaders, this invariant is enforced across leaders by making use of the disjointness invariant—a leader must only assign timestamps within the interval of its leader lease
pp.7 LHS, Spanner also enforces the following **external-consistency invariant**: if the start of a transaction T_2 occurs after the commit of a transaction T_1 then the commit timestamp of T_2 must be greater than the commit timestamp of T_1, the protocol for executing transactions and assigning timestamps obeys two rules which together guarantee this invariant

- **start:** the coordinator leader for a write T_i assigns a commit timestamp s_i no less than the value of $TT.now().latest$
- **commit wait:** the coordinator leader ensures that clients cannot see any data committed by T_i until $TT.after(s_i)$ is true

serving reads at a timestamp: pp.7 LHS, the monotonicity invariant described above allows Spanner to correctly determine whether a replica's state is sufficiently up-to-date to satisfy a read, every replica tracks a value called *safe time* t_{safe} which is the maximum timestamp at which a replica is up-to-date, a replica can satisfy a read at a timestamp t if $t \leq t_{safe}$
pp.7 RHS, define $t_{safe} = \min(t_{safe}^{Paxos}, t_{safe}^{TM})$ where each Paxos state machine has a safe time t_{safe}^{Paxos} and each transaction manager has a safe time t_{safe}^{TM}, t_{safe}^{Paxos} is simpler—it is the timestamp of the highest-applied Paxos write, $t_{safe}^{TM} = \infty$ at a replica if there are zero prepared but not committed transactions, $t_{safe}^{TM} = \min_i(s_{i,g}^{prepare}) - 1$ over all transactions T_i prepared at participant group g

assigning timestamps to RO transactions: pp.7 RHS, a read-only transaction executes in two phases—assign a timestamp s_{read} and then execute the transaction's reads as snapshot reads at s_{read}, the snapshot reads can execute at any replicas that are sufficiently up-to-date, the simple assignment of $s_{read} = TT.now().latest$ at any time after a transaction starts preserves external consistency, to reduce the chances of blocking Spanner should assign the oldest timestamp that preserves external consistency

10.5.2 Details

read-write transactions: pp.8 LHS, like Bigtable writes that occur in a transaction are buffered at the client until commit, as a result reads in a transaction do not see the effects of the transaction's writes, reads within read-write transactions use would-wait to avoid deadlocks, when a client has completed all reads and buffered all writes it begins two-phase commit, having the client drive two-phase commit avoids sending data twice across wide-area links
pp.8 LHS, a non-coordinator-participant leader first acquires write locks, it then chooses a prepare timestamp that must be larger than any timestamps it has assigned to previous transactions, each participant then notifies the coordinator of its prepare timestamp, the coordinator leader also first acquires write locks but skip the prepare phase, it chooses a timestamp for the entire transaction after hearing from all other participant leaders, the commit timestamp s must be greater or equal to all prepare timestamps, the coordinator leader then logs a commit record through Paxos, before

allowing any coordinator replica to apply the commit record the coordinator leader waits until $TT.after(s)$ so as to obey the commit-wait rule, after commit wait the coordinator sends the commit timestamp to the client and all other participant leaders, each participant leaders logs the transaction's outcome through Paxos, all participants apply at the same timestamp and then release locks

read-only transactions: pp.8 RHS, Spanner requires a *scope* expression for every read-only transaction, which is an expression that summarizes the keys that will be read by the entire transaction, Spanner automatically infers the scope for standalone queries, if the scope's values are served by a single Paxos group then the client issues the read-only transaction to that group's leader, that leader assigns s_{read} and executes the read, define $LastTS()$ to be the timestamp of the last committed write at a Paxos group, if there are no prepared transactions the assignment $s_{read} = LastTS()$ trivially satisfies external consistency, if the scope's values are served by multiple Paxos groups there are several options, the most complicated option is to do a round of communication with all of the groups' leaders to negotiate s_{read} based on $LastTS()$, Spanner currently implements a simpler choice—the client avoids a negotiation round and just has its reads execute at $s_{read} = TT.now().latest$

schema-change transactions: pp.8 RHS, a Spanner schema-change transaction is a generally non-blocking variant of a standard transaction, first it is explicitly assigned a timestamp in the future which is registered in the prepare phase, second reads and writes synchronize with any registered schema-change timestamp at time t—they may proceed if their timestamps precede t, but they must block behind the schema-change transaction if their timestamps are after t

refinement: pp.9 LHS, there are three

- t_{safe}^{TM} has a weakness in that a single prepared transaction prevents t_{safe} from advancing, as a result no reads can occur at later timestamps even if the reads do not conflict with the transaction, such false conflicts can be removed by augmenting t_{safe}^{TM} with a fine-grained mapping from key ranges to prepared transaction timestamps, when a read arrives it only needs to be checked against the fine-grained safe time for key ranges with which the read conflicts
- $LastTS()$ has a similar weakness—if a transaction has just committed a non-conflicting read-only transaction must still be assigned s_{read} so as to follow that transaction, this weakness can be remedied similarly by augmenting $LastTS()$ with a fine-grained mapping from key ranges to commit timestamps in the lock table, when a read-only transaction arrives its timestamp can be assigned by taking the maximum value of $LastTS()$ for the key ranges with which the transaction conflicts
- t_{safe}^{Paxos} has a weakness in that a snapshot read at t cannot execute at Paxos groups whose last write happened before t, Spanner addresses this problem by taking advantage of the disjointness of leader-lease intervals, each Paxos leader advances t_{safe}^{Paxos} by keeping a threshold above which future writes' timestamp will occur—it maintains a mapping $MinNextTS(n)$ from Paxos sequence number n to the minimum timestamp that may be assigned to Paxos sequence number $n + 1$, a replica can advance t_{safe}^{Paxos} to $MinNextTS(n) - 1$ when it has applied through n, the disjointness invariant enforces $MinNextTS()$ promises across leaders

10.6 Evaluation

microbenchmarks: pp.10, snapshot reads can execute at any up-to-date replicas so their throughput increases almost linearly with the number of replicas

single-read read-only transactions only execute at leaders because timestamp assignment must happen at leaders, read-only transaction throughput increases with the number of replicas because the number of effective spanservers increases
write throughput benefits from the same experimental artifact, but that benefit is outweighed by the linear increase in the amount of work performed per write as the number of replicas increases

availability: pp.10 RHS, shorter lease times would reduce the effect of server deaths on availability, but would require greater amounts of lease-renewal network traffic

F1: pp.11 RHS, the F1 team chose to use Spanner for several reasons: (1) Spanner removes the need to manually reshard (2) Spanner provides synchronous replication and automatic failover (3) F1 requires strong transactional semantics which made using other NoSQL systems impractical

10.7 References

Marzullo's algorithm: Keith Marzullo and Susan Owicki, *Maintaining the time in a distributed system*, Proc. of PODC. 1983, pp.295—305

wound-wait: Daniel Peng and Frank Dabek, *Large-scale incremental processing using distributed transactions and notifications*, Proc. of OSDI. 2010, pp.1—15

11 Lesson #10 Scaling Memcache at Facebook

11.1 The Publication Details

Author: Rajesh Nishtala et al.

Title: Scaling Memcache at Facebook

Journal: 10th USENIX Symposium on Networked Systems Design and Implementation, pp.385

DOI: `https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf`

11.2 Overview

pp.1 LHS, Facebook leverages *memcached* as a building block to construct and scale a distributed key-value store that supports the world's largest social network; pp.2 LHS, as is *memcached* is an in-memory hash table running on a single server; pp.1 RHS, it provides a simple set of operations (`set`, `get`, and `delete`) that makes it attractive as an elemental component in a large-scale distributed system, the following properties greatly influence

- users consume an order of magnitude more content than they create, this behavior results in a workload dominated by fetching data and suggests that caching can have significant advantages
- our read operations fetch data from a variety of sources such as MySQL databases, HDFS installations, and backend servers, this heterogeneity requires a flexible caching strategy able to store data from disparate sources

in the following we use "*memcached*" to refer to the source code or a running binary and "*memcache*" to describe the distributed system

query cache: we use *memcache* as a demand-filled look-aside cache, for write requests the web server issues SQL statements to the database and then sends a delete request to *memcache* that invalidates any stale data, we choose to delete cached data instead of updating it because deletes are idempotent

generic cache: we also leverage *memcache* as a more general key-value store, e.g. engineers use *memcache* to store pre-computed results from sophisticated machine learning algorithms which can then be used by a variety of other applications

11.3 In a Cluster: Latency and Load

11.3.1 Reducing latency

pp.3 LHS, items are distributed across the *memcached* servers through consistent hashing, thus web servers have to routinely communicate with many *memcached* servers to satisfy a user request, data replication often alleviates the single-server bottleneck but leads to significant memory inefficiencies in the common case

parallel requests and batching: pp.3 LHS, we construct a directed acyclic graph (DAG) representing the dependencies between data, a web server uses this DAG to maximize the number of items that can be fetched concurrently

client-server communication: pp.3 LHS, we embed complexity of the system into a stateless client rather than in the *memcached* servers, keeping the clients stateless enables rapid iteration in the software and simplifies our deployment process, client logic is provided as two components—a library that can be embedded into applications or as a standalone proxy named *mcrouter*, this proxy presents a *memcached* server interface and routes the requests/replies
pp.3 LHS, we rely on UDP for get requests to reduce latency and overhead, since UDP is connectionless, each thread in the web server is allowed to directly communicate with *memcached* servers directly, bypassing *mcrouter*, without establishing and maintaining a connection thereby reducing the overhead, the UDP implementation detects packets that are dropped or received out of order (using sequence numbers) and treats them as errors on the client side, it does not provide any mechanism to try to recover from them
pp.3 RHS, for reliability clients perform set and delete operations over TCP through an instance of *mcrouter* running on the same machine as the web server, for operations where we need to confirm a state change (updates and deletes) TCP alleviates the need to add a retry mechanism to our UDP implementation, web servers rely on a high degree of parallelism and over-subscription to achieve high throughput, the high memory demands of open TCP connections makes it prohibitively expensive to have an open connection between every web thread and *memcached* server without some form of connection coalescing via *mcrouter*

incast congestion: pp.4 LHS, clients use a sliding window mechanism to control the number of outstanding requests, when the client receives a response the next request can be sent, similar to TCP's congestion control the size of this sliding window grows slowly upon a successful request and shrinks when a request goes unanswered, the window applies to all *memcache* requests independently of destination, whereas TCP windows apply only to a single stream
pp.4 LHS, with lower window sizes the application will have to dispatch more groups of *memcache* requests serially increasing the duration of the web request, as the window size gets too large the number of simultaneous *memcache* requests causes incast congestion, the result will be *memcache* errors and the application falling back to the persistent storage for the data

11.3.2 Reducing load

pp.4 RHS, we use *memcache* to reduce the frequency of fetching data along more expensive paths such as database queries, web servers fall back to these paths when the desired data is not cached

leases: pp.4 RHS, we introduce a new mechanism called *leases* to address two problems—stale sets and thundering herds

stale set: a stale set occurs when a web server sets a value in *memcache* that does not reflect the latest value that should be cached, this can occur when concurrent updates to *memcache* get reordered, a *memcached* instance gives a lease to a client to set data back into the cache when that client experiences a cache miss, the lease is a 64-bit token bound to the specific key the client originally requested, when the lease token *memcached* can verify and determine whether the data should be stored and thus arbitrate concurrent writes, verification can fail if *memcached* has invalidated the lease token due to receiving a delete request for that item, leases prevent stale sets in a manner similar to how load-link/store-conditional operates
with leases we can minimize the aplication's wait time, we can further reduce time by identifying situations in which returning slightly out-of-date data is acceptable, when a key is deleted its value is transferred to a data structure that holds recently deleted items, where it lives for a short time before being flushed, a get request can return a lease token or data that is marked as stale

thundering herd: a thundering herd happens when a specific key undergoes heavy read and write activity, as the write activity repeatedly invalidates the recently set values, many reads default to the more costly path, each *memcached* server regulates the rate at which it returns tokens, by default we configure these servers to return a token only once every 10 seconds per key, requests for a key's value within 10 seconds of a token being issued results in a special notification telling the client to wait a short amount of time

***memcache* pools:** pp.5 LHS, different applications' workloads can produce negative interference resulting in decreased hit rates, to accommodate these differences we partition a cluster's *memcached* servers into separate pools, we designate one pool named *wildcard* as the default and provision separate pools for keys whose residence in wildcard is problematic

replication within pools: pp.5 RHS, within some pools we use replication to improve the latency and efficiency of *memcached* servers, we choose to replicate a category of keys within a pool when (1) the application routinely fetches many keys simultaneously (2) the entire data set fits in one or two *memcached* servers (3) the request rate is much higher than what a single server can manage, we favor replication in this instance over further dividing the key space, this approach requires delivering invalidations to all replicas to maintain consistency

11.3.3 Handling failures

pp.5 RHS, there are two scales at which we must address failures: (1) a small number of hosts are inaccessible due to a network or server failure (2) a widespread outage that affects a significant percentage of the servers within the cluster, for small outages we rely on an automated remediation system, to further insulate backend services from failure we dedicate a small set of machines named *Gutter* to take over the responsibilities of a few failed servers, Gutter accounts for approximately 1% of the *memcached* servers in a cluster
pp.5 RHS, when a *memcached* client receives no response to its get request the client assumes the server has failed and issues the request again to a special Gutter pool, if this second request misses the client will insert the appropriate key-value pair into the Gutter machine after querying the database, entries

in Gutter expire quickly to obviate Gutter invalidations, Gutter limits the load on backend services at the cost of slightly stale data, ordinarily each failed request results in a hit on the backing store potentially overloading it, by using Gutter to store these results a substantial fraction of these failures are converted into hits in the gutter pool thereby reducing load on the backing store

11.4 In a Region: Replication

pp.6 LHS, we split our web and *memcached* servers into multiple *frontend clusters*, these clusters along with a storage cluster that contain the databases define a *region*, we trade replication of data for more independent failure domains, tractable network configuration, and a reduction of incast congestion

11.4.1 Regional invalidations

pp.6 LHS, the storage cluster is responsible for invalidating cached data to keep frontend clusters consistent with the authoritative versions, SQL statements that modify authoritative state are amended to include *memcache* keys that need to be invalidated once the transaction commits, we deploy invalidation daemons named *mcsqueal* on every database

reducing packet rates: pp.6 RHS, invalidation daemons batch deletes into fewer packets, the batching results in an 18-fold improvement in the median number of deletes per packet

invalidation via web servers: pp.6 RHS, it is simpler for web servers to broadcast invalidations to all frontend clusters, this approach unfortunately suffers from two problems—first it incurs more packet overhead as web servers are less effective at batching invalidations than *mcsqueal* pipeline, second it provides little recourse when a systemic invalidation problem arises such as misrouting of deletes due to a configuration error, in the past this would often require a rolling restart of the entire *memcache* infrastructure, in contrast embedding invalidations in SQL statements allows *mcsqueal* to simply replay invalidations that may have been lost or misrouted

11.4.2 Regional pools

pp.7 LHS, if users' requests are randomly routed to all available frontend clusters then the cached data will be roughly the same across all the frontend clusters, this allows us to take a cluster offline for maintenance without suffering from reduced hit rates, we can reduce the number of replicas by having multiple frontend clusters share the same set of *memcached* servers, we call this a *regional pool*
pp.7 LHS, replication trades more *memcached* servers for less inter-cluster bandwidth, lower latency, and better fault tolerance, one of the main challenges of scaling *memcache* within a region is deciding whether a key needs to be replicated across all frontend clusters or have a single replica per region

11.4.3 Cold cluster warmup

pp.7 RHS, a system called *cold cluster warmup* mitigates poor hit rate of new clusters by allowing clients in the "cold cluster" (i.e. the frontend cluster that has an empty cache) to retrieve data from the "warm cluster" (i.e. a cluster that has caches with normal hit rates) rather than the persistent storage

11.5 Across Regions: Consistency

pp.7 RHS, each region consists of a storage cluster and several frontend clusters, we designate one region to hold the master databases and the other regions to contain read-only replicas, we rely on MySQL's replication mechanism to keep replica databases up-to-date with their masters

writes from a master region: pp.8 LHS, requiring the storage cluster to invalidate data via daemons avoids a race condition in which an invalidation arrives before the data has been replicated from the master region

writes from a non-master region: pp.8 LHS, a cache refill from a replica's database should only be allowed after the replication stream has caught up, we employ a *remote marker* mechanism to minimize the probability of reading stale data, we implement remote markers by using a regional pool, the presence of the marker indicates that data in the local replica database are potentially stale and the query should be redirected to the master region, we explicitly trade additional latency when there is a cache miss for a decreased probability of reading stale data

operational considerations: pp.8 RHS, by sharing the same channel of communication for the delete stream as the database replication we gain network efficiency on lower bandwidth connections
the aforementioned system for managing deletes is also deployed with the replica databases to broadcast the deletes to *memcached* servers in the replica regions, databases and *mcrouter*s buffer deletes when downstream components become unresponsive, a failure or delay in any of the components results in an increased probability of reading stale data, the buffered deletes are replayed once these downstream components are available again

11.6 Single Server Improvements

11.6.1 Performance optimizations

pp.9 LHS, the first major optimizations were to (1) allow automatic expansion of the hash table to avoid look-up times drifting to $O(n)$ (2) make the server multi-threaded using a global lock to protect multiple data structures (3) giving each thread its own UDP port to reduce contention when sending replies and later spreading interrupt processing overhead

11.6.2 Adaptive slab allocator

pp.9 RHS, *memcached* employs a slab allocator to manage memory, the allocator organizes memory into *slab classes* each of which contains pre-allocated uniformly sized chunks of memory, *memcached* stores items in the smallest possible slab class that can fit the item's metadata, key, and value, slab classes start at 64B and exponentially increase in size by a factor of 1.07 up to 1MB aligned on 4B boundaries, once a *memcached* server can no longer allocate free memory, storage for new items is done by evicting the least recently used (LRU) item within that slab class
pp.10 LHS, we implemented an adaptive allocator, which identifies slab classes as needing more memory if they are currently evicting items and if the next item to be evicted as used at least 20% more recently than the average of the least recently used items in other slab classes, our algorithm focuses on balancing the age of the oldest items among classes, balancing age provides a better approximation to a single global LRU eviction policy for the entire server rather than adjusting eviction rates which can be heavily influenced by access patterns

11.6.3 The transient item cache

pp.10 LHS, while *memcached* supports expiration times entries may live in memory well after they have expired, we introduce a hybrid scheme that relies on lazy eviction for most keys and proactively evicts short-lived keys when they expire, we place short-lived items into a circular buffer of linked lists called the *transient item cache* based on the expiration time of the item, every second all of the items in the bucket at the head of the buffer are evicted and the head advances by one

11.6.4 Software upgrades

pp.10 LHS, we modified *memcached* to store its cached values and main data structures in System V shared memory regions so that the data can remain live across a software upgrade

11.7 Memcache Workload

11.7.1 Pool statistics

pp.11 LHS, we now discuss key metrics of four *memcache* pools, the pools are wildcard (the default pool), app (a pool devoted for a specific application), a replicated pool for frequently accessed data, and a regional pool for rarely accessed information

11.7.2 Invalidation latency

pp.11 LHS, we recognize that the timeliness of invalidations is a critical factor in determining the probability of exposing stale data

11.8 Related Work

pp.12 LHS, *memcache* does not guarantee persistence, we rely on other systems to handle persistent data storage

11.9 Conclusion

pp.12 RHS, while building, maintaining, and evolving our system we have learned the following

- separating cache and persistent storage systems allows us to independently scale them
- managing stateful components is operationally more complex than stateless ones, as a result keeping logic in a stateless client helps iterate on features and minimize disruption
- the system must support gradual rollout and rollback of new features even if it leads to temporary heterogeneity of feature sets

11.10 References

Little's law: LITTLE, J., AND GRAVES, S. Little's law. Building Intuition (2008), pp.81—100

12 Lesson #10 Scalable Causal Consistency for Wide-Area Storage with COPS

12.1 The Publication Details

Author: Wyatt Lloyd et al.

Title: Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS

Journal: SOSP 11, October 2011, pp.401—416

DOI: `http://istc-cc.cmu.edu/publications/papers/2011/cops-sosp2011.pdf`

12.2 Abstract

pp.1 LHS, geo-replicated distributed data stores that support complex online applications must provide an "always-on" experience, today's systems often sacrifice strong consistency to achieve these goals, in this paper we identify and define a consistency model, causal consistency with convergent conflict handling or causal+, that is the strongest achieved under these constraints

12.3 Introduction

pp.1 RHS, modern web services have chosen overwhelmingly to embrace availability and partition tolerance at the cost of strong consistency, this choice also enables these systems to provide low latency for client operations and high scalability, we refer to systems with these four properties—**A**vailiabity, low **L**atency, **P**artition-tolerance, and high **S**calability—as ALPS systems, given that ALPS systems must sacrifice strong consistency (i.e. linearizability) we seek the strongest consistency model that is achievable under these constraints, in this paper we consider causal consistency with convergent conflict handling which we refer to as causal+ consistency, the causal component of causal+ consistency ensures that the data store respects the causal dependencies between operations, the convergent conflict handling component of causal+ consistency ensures that replicas never permanently diverge and that conflicting updates to the same key are dealt with identically at all sites, when combined with causal consistency this property ensures that clients see only progressively newer versions of keys
pp.2 LHS, we detail two versions of our COPS (Clusters of Order-Preserving Servers) system

COPS: the regular version provides scalable causal+ consistency between individual items in the data store even if their causal dependencies are spread across many different machines in the local datacenter, the scalability requirements for ALPS systems creates the largest distinction between COPS and prior causal+ consistent systems, the performance and overhead of COPS is similar to prior systems

COPS-GT: the extended version provides get transactions that give clients a consistent view of multiple keys, our get transactions require no locks, are non-blocking, and take at most two parallel rounds of intra-datacenter requests, to the best of our knowledge COPS-GT is the first ALPS system to achieve non-blocking scalable get transactions, compared to the regular version COPS-GT is less efficient for certain workloads (e.g. write-heavy) and is less robust to long network partitions and datacenter failures

12.4 ALPS Systems and Trade-offs

pp.2 RHS, a distributed storage system has multiple sometimes competing goals which include

stronger consistency: an ideal data store would provide linearizability, sometimes informally called strong consistency, which dictates that operations appear to take effect across the entire system at a single instance in time between the invocation and completion of the operation

partition tolerance: the data store continues to operate under network partitions

availability: all operations issued to the data store complete successfully, no operation can block indefinitely or return an error signifying that data is unavailable

low latency: client operations complete quickly

high scalability: the data store scales out linearly—adding N resources to the system increases aggregate throughput and storage capacity by $O(N)$

the CAP theorem proves that a shared-data system that has availability and partition tolerance cannot achieve linearizability, low latency—defined as latency less than the maximum wide-area delay between replicas—has also been proven incompatible with linearizability and sequential consistency

12.5 Causal+ Consistency

pp.3 LHS, we restrict our consideration to a key-value data store with two basic operations: put(key,val) and get(key) = val, these are equivalent to write and read operations in a shared-memory system, values are stored and retrieved from logical *replicas*, in our COPS system a single logical replica corresponds to an entire local cluster of nodes
pp.3 LHS, an important concept in our model is the notion of *potential causality* between operations, three rules define potential causality denoted by $\rightsquigarrow$

execution thread: if a and b are two operations in a single thread of execution, then $a \rightsquigarrow b$ if operation a happens before operation b

gets from: if a is a put operation and b is a get operation that returns the value written by a, then $a \rightsquigarrow b$

transitivity: for operations a, b, and c, if $a \rightsquigarrow b$ and $b \rightsquigarrow c$ then $a \rightsquigarrow c$

like many our model does not allow threads to communicate directly, requiring instead that all communication occur through the data store

12.5.1 Definition

pp.3 LHS, we define causal+ consistency as a combination of two properties—causal consistency and convergent conflict handling

causal consistency: requires that values returned from get operations at a replica are consistent with the other defined by $\rightsquigarrow$, causal consistency does not order concurrent operations, if $a \not\rightsquigarrow b$ and $b \not\rightsquigarrow a$ then a and b are concurrent

convergent conflict handling: requires that all conflicting puts be handled in the same manner at all replicas using a handler function h, this handler function h must be associative and commutative so that replicas can handle conflicting writes in the order they receive them and that the results of these handlings will converge, e.g. one replica's $h(a, h(b, c))$ = another's $h(c, h(b, a))$
one common way to handle conflicting writes in a convergent fashion is the last-writer-wins rule (also called Thomas's write rule), which declares one of the conflicting writes as having occurred later and has it overwrite the earlier write, the default version of COPS uses the last-write-wins rule

12.5.2 Causal+ vs. other consistency models

pp.3 RHS, the distributed systems literature defines several popular consistency models, in decreasing strength they include

strong consistency: maintains a global real-time ordering

sequential consistency: ensures at least a global ordering

causal consistency: ensures partial orderings between dependent operations

FIFO (PRAM) consistency: only preserves the partial ordering of an execution thread, not between threads

per-key sequential consistency: ensures all operations for each individual key have a global order

eventual consistency: a "catch-all" term used suggesting eventual convergence to some type of agreement

the causal+ consistency we introduce falls between sequential and causal consistency, it is weaker than sequential consistency but sequential consistency is provably not achievable in an ALPS system, it is stronger than causal consistency and per-key sequential consistency and it is achievable for ALPS systems

12.5.3 Causal+ in COPS

pp.4 LHS, we use two abstractions in the COPS system—version and dependencies

version: we refer to the different values a key has as the versions of a key which we denote by $\text{key}_{\text{version}}$, in COPS versions are assigned in a manner that ensures that if $x_i \rightsquigarrow y_j$ then $i < j$, each replica in COPS always returns non-decreasing versions of a key, we refer to this as causal+ consistency's **progressing property**

dependencies: we say y_j depends on x_i if and only if $\mathsf{put}(x_i) \rightsquigarrow \mathsf{put}(y_j)$, COPS provides causal+ consistency during replication by writing a version only after writing all of its dependencies

12.5.4 Scalable causality

pp.4 RHS, log-exchange-based serialization inhibits replica scalability as it relies on a single serialization point in each replica to establish ordering, COPS achieves scalability by explicitly encoding dependencies in metadata associated with each key's version, when keys are replicated remotely the receiving datacenter performs dependency checks before committing the incoming version

12.6 System Design of COPS

pp.4 RHS, there are two distinct versions of the system—the first which we refer to simply as COPS provides a data store that is causal+ consistent, the second called COPS-GT provides a superset of this functionality by also introducing support for get transactions, because of the additional metadata needed to enforce the consistency properties of get transactions, a given deployment must run exclusively as COPS or COPS-GT

12.6.1 Overview of COPS

pp.4 RHS, COPS is a key-value storage system designed to run across a small number of datacenters, each datacenter has a local COPS cluster with a complete replica of the stored data, a client of COPS is an application that uses the COPS client library to call directly into the COPS key-value store, clients communicate only with their local COPS cluster running in the same datacenter, each local COPS cluster is set up as a linearizable key-value store, linearizable systems can be implemented scalably by partitioning the keyspace into N linearizable partitions and having clients access each partition independently, the composability of linearizability ensures that the resulting system as a whole remains linearizable, linearizability is acceptable locally because we expect very low latency and no partitions within a cluster, on the other hand replication between COPS clusters happens asynchronously to ensure low latency for client operations and availability in the face of external partitions
pp.5 LHS, COPS is composed of two main software components

key-value store: the basic building block in COPS is a standard key-value store that provides linearizable operations on keys, COPS extends the standard key-value store in two ways and COPS-GT adds a third extension

1. each key-value pair has associated metadata, in COPS this metadata is a version number, in COPS-GT it is both a version number and a list of dependencies (other keys and their respective versions)
2. the key-value store exports three additional operations as part of its key-value interface: get_by_version, put_after, and dep_check which enable the COPS client library and an asynchronous replication process that supports causal+ consistency and get transaction
3. for COPS-GT the system keeps around old versions of key-value pairs not just the most recent put to ensure that it can provide get transactions

client library: pp.5 LHS, the client library exports two main operations to applications—reads via get in COPS or get_trans in COPS-GT and writes via put, the client library also maintains state about a client's current dependencies through a context parameter in the client library API

pp.5 LHS, the COPS design strives to provide causal+ consistency with resource and performance overhead similar to existing eventually consistent systems, COPS and COPS-GT must therefore

- minimize overhead of consistency-preserving replication—we present a mechanism that requires only a small number of such checks by leveraging the graph structure inherent to causal dependencies
- minimize space requirement—COPS-GT uses aggressive garbage collection to prune old state
- ensure fast get_trans operations—we present an algorithm for get_trans that completes in at most two rounds of local get_by_version operations

12.6.2 The COPS key-value store

pp.5 RHS, in COPS the system stores the most recent version number and value for each key, in COPS-GT the system maps each key to a list of version entries each consisting of $\langle version, value, deps \rangle$, the *deps* field is a list of the version's zero or more dependencies, each dependency is a $\langle key, version \rangle$ pair, each COPS cluster maintains its own copy of the key-value store, for scalability our implementation partitions the keyspace across a cluster's nodes using consistent hashing, for fault tolerance each key is replicated across a small number of nodes using chain replication, gets and puts are linearizable across the nodes in the cluster, every key stored in COPS has one *primary node* in each cluster, we term the set of primary nodes for a key across all clusters as the *equivalent nodes* for that key, COPS's consistent hashing assigns each node responsibility for a few different key ranges, after a write completes locally the primary node places it in a replication queue from which it is sent asynchronously to remote equivalent nodes, those nodes in turn wait until the value's dependencies are satisfied in their local clusters before locally committing the value, this dependency checking mechanism ensures writes happen in a causally consistent order and reads never block

12.6.3 Client library and interface

pp.6 LHS, besides put and get (get_trans for COPS-GT) the client API provides createContext and deleteContext operations, the client API differs from a traditional key-value interface in two ways—first COPS-GT provides get_trans which returns a consistent view of multiple key-value pairs in a single call, second all functions take a context argument *ctx_id* which the library uses internally to track causal dependencies across each client's operations, the context defines the causal+ "thread of execution",

by separating different threads of execution COPS avoids false dependencies that would result from intermixing them

COPS-GT client library: pp.6 LHS, the client library in COPS-GT stores the client's context in a table of $\langle key, version, deps \rangle$ entries, when a client gets a key from the data store the library adds this key and its causal dependencies to the context, when a client puts a value the library sets the put's dependencies to the most recent version of each key in the current context, a successful put into the data store returns the version number v assigned to the written value, the client library then adds this new entry $\langle key, v, D \rangle$ to the context, the context therefore includes all values previously read or written in the client's session as well as all of those dependencies' dependencies
pp.6 RHS, to mitigate the client and data store state required to track dependencies COPS-GT provides garbage collection that removes dependencies once they are committed to all COPS replicas, we term dependencies that must be checked the *nearest dependencies*, which are sufficient for the key-value store to provide causal+ consistency, the full dependency list is only needed to provide get_trans operations in COPS-GT

COPS client library: pp.6 RHS, the COPS client library stores only $\langle key, version \rangle$ entries, for a get operation the retrieved $\langle key, value \rangle$ is added to the context, for a put operation the library uses the current context as the nearest dependencies, clears the context, and then repopulates it with only this put

12.6.4 Writing values in COPS and COPS-GT

pp.6 RHS, all writes in COPS first go to the client's local cluster and then propagate asynchronously to remote clusters, the key-value store exports a single API call to provide both operations

writes to the local cluster: when a client calls put(key, val, ctx_id) the library computes the complete set of dependencies *deps* and identifies some of those dependency tuples as the value's nearest ones, the library then calls put_after without the version argument, the put_after operation ensures that *val* is committed to each cluster only after all of the entries in its dependency list have been written, in the client's local cluster this property holds automatically as the local store provides linearizability, as the primary node uses a Lamport timestamp to assign a unique version number to each update, which allow COPS to derive a single global order over all writes for each key, this order implicitly implements the last-writer-wins convergent conflict handling policy

write replication between clusters: after a write commits locally the primary storage node asynchronously replicates that write to its equivalent nodes in different clusters using a stream of put_after operations, here however the primary node includes the key's version number in the put_after call, this approach requires the remote nodes receiving updates to commit an update only after its dependencies have been committed to the same cluster, to ensure this property a node that receives a put_after request from another cluster must determine if the value's nearest dependencies have already been satisfied locally, it does so by issuing a dep_check, when a node receives a dep_check it examines its local state to determine if the dependency value has already been written, if so it immediately responds to the operation, if not it blocks until the needed version has been written, if all dep_check operations on the nearest dependencies succeed the node hanlding the put_after request commits the written value, if any dep_check operation times out the node handling the put_after reissue it, potentially to a new node if a failure occurred

12.6.5 Reading values in COPS

pp.7 RHS, like writes reads are satisfied in the local cluster, requesting the latest version is equivalent to a regular single-key get, requesting a specific version is necessary to enable get transactions, accordingly get_by_version operations in COPS always request the latest version

12.6.6 Get transactions in COPS-GT

pp.7 RHS, to retrieve multiple values in a causal+ consistent manner a client calls get_trans with the desired set of keys, the COPS client library implements the get transactions algorithm in two rounds

round I: the library issues n concurrent get_by_version operations to the local cluster, one for each key the client listed in get_trans, each get_by_version operation returns a $\langle value, version, deps \rangle$ tuple where *deps* is a list of keys and versions, the client library then examines every dependency entry $\langle key, version \rangle$, the causal dependencies for that result are satisfied if either the client did not request the dependent key, or if it did the version it retrieved was $\geq$ the version in the dependency list, for all keys that are not satisfied the library issues a second round of concurrent get_by_version operations, the version requested will be the newest version seen in any dependency list from the first round

round II: happens only when the client must read newer versions than those retrieved in the first round, this case occurs only if keys involved in the get transaction are updated during the first round, thus we expect the second round to be rare, the causal+ consistency of the data store provides two important properties for the get transaction algorithm's second round—first the get_by_version requests will succeed immediately as the requested version must already exist in the local cluster, second the new get_by_version requests will not introduce any new dependencies as those dependencies were already known in the first round due to transitivity, which is the resaon why the get transaction algorithm specifies an explicit version in its second round rather than just getting the latest

12.7 Garbage, Faults, and Conflicts

12.7.1 Garbage collection subsystem

version garbage collection: pp.8 RHS, COPS-GT only, COPS-GT limits the total running time of get_trans through a configurable parameter *trans_time*, after a new version of a key is written COPS-GT only needs to keep the old version around for trans_time plus a small delta for clock skew, after this time no get_by_version call will subsequently request the old version and the garbage collection can remove it, thus the space overhead is bounded by the number of old versions that can be created within the *trans_time*

dependency garbage collection: pp.8 RHS, COPS-GT only, after *trans_time* seconds after a value has been committed in all datacenters, COPS-GT can clean a value's dependencies, to clean dependencies each remote datacenter notifies the originating datacenter when the write has committed and the timeout period has elapsed, once all datacenters confirm the originating datacenter cleans its own dependencies and informs the others to do likewise, dependencies are only collected on the most recent version of the key, during a partition dependencies on the most recent versions of keys cannot be collected, this is a limitation of COPS-GT although we expect long partitions to be rare

client metadata garbage collection: pp.9 LHS, COPS and COPS-GT, the COPS client library tracks all operations during a client session using the *ctx_id* passed with all operation, in contrast to the dependency information which resides in the key-value store itself, the dependencies

discussed here are part of the client metadata and are stored in the client library, COPS reduces the size of this client state in two ways

1. once a put_after commits successfully to all datacenters COPS flags that key version as *never-depend*, get_by_version results include this flag and the client library will immediately remove a never-depend item from the list of dependencies in the client context, furthermore this process is transitive—anything that a never-depend key depended on must have been flagged never-depend, so it too can be garbage collected from the context
2. when a node receives a put_after it checks each item in the dependency list and removes items with version numbers older than a global checkpoint time, which is the newest Lamport timestamp that is satisfied at all nodes across the entire system, the COPS key-value store returns this checkpoint time to the client library allowing the library to clean these dependencies from the context

12.7.2 Fault tolerance

pp.9 RHS, we assume that failures are fail-stop

client failures: COPS's key-value interface means that each client request through library is handled independently and atomically by the data store, from the storage system's perspective if a client fails it simply stops issuing new requests, no recovery is necessary, from a client's perspective COPS's dependency tracking makes it easier to handle failures of other clients

key-value node failures: we built our system on top of independent clusters of FAWN-KV nodes which use chain replication within a cluster to mask node failures, similar to the design of FAWN-KV each data item is stored in a chain of R consecutive nodes along the consistent hashing ring, put_after operations are sent to the head of the appropriate chain, propagate along the chain, and then commit at the tail, which then acknowledges the operation, get_by_version operations are sent to the tail which responds directly

datacenter failures: first any put_after operations that originated in the failed datacenter but which were not yet copied out will be lost
second the storage required for replication queues in the active datacenters will grow, as they will be unable to send put_after operations to the failed datacenter and thus COPS will be unable to garbage collect those dependencies
third in COPS-GT dependency garbage collection cannot continue in the face of a datacenter failure until either the partition is healed or the system is reconfigured to exclude the failed datacenter

12.7.3 Conflict detection

pp.10 LHS, conflicts occur when there are two simultaneous writes to a given key, the default COPS system avoids conflict detection using a last-writer-wins strategy, the last write is determined by comparing version numbers, COPS with conflict detection or COPS-CD adds three new components to the system

1. all put operations carry with them previous version metadata, which indicates the most recent previous version of the key that was visible at the local cluster at the time of the write
2. all put operations now have an implicit dependency on that previous version, which ensures that a new version will only be written after its previous version
3. COPS-CD has an application-specified convergent conflict handler that is invoked when a conflict is detected

12.8 Evaluation

12.8.1 Dynamic workloads

pp.11 RHS, as the inter-operation delay increases the number of dependencies per operation decreases because of the ongoing garbage collection; pp.13 LHS, as processing time per operation increases the inter-operation delay correspondingly increases, which in turn leads to fewer dependencies
pp.13 LHS, the throughput of COPS is unaffected by different variances, get operations in COPS never inherit extra dependencies as the returned value is always nearer by definition, however COPS-GT has an increased chance of inheriting dependencies as variance increases which results in decreased throughput

12.8.2 Scalability

pp.13 RHS, this workload has a higher inter-operation delay, larger values, and a read-heavy distribution, under these settings the throughput of COPS and COPS-GT are very comparable and both scale well with the number of servers

12.9 Conclusion

pp.14 RHS, COPS achieves causal+ consistency by tracking and explicitly checking that causal dependencies are satisfied before exposing writes in each cluster, COPS-GT builds upon COPS by introducing get transactions that enable clients to obtain a consistent view of multiple keys

12.10 References

Thomas's write rule: R. H. Thomas, *A majority consensus approach to concurrency control for multiple copy databases*, ACM Trans. Database Sys., 4(2), 1979

13 Lesson #11 A Scalable Peer-to-Peer Lookup Service for Internet Applications

13.1 The Publication Details

Author: Ion Stoica et al.

Title: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications

Journal: ACM SIGCOMM 01, Vol. 31, Issue 4, October 2001, pp.149—160

DOI: `https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf`

13.2 Introduction

pp.1 LHS, peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization where the software running at each node is equivalent in functionality, the core operation in most peer-to-peer systems is efficient location of data items, the contribution of this paper is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures, the Chord protocol supports just one operation—given a key it maps the key onto a node, Chord uses a variant of consistent hashing to assign keys to Chord nodes; pp.1 RHS, in the steady state in an N-node system each node maintains information only about $O(\log N)$ other nodes and

resolves all lookups via $O(\log N)$ messages to other nodes, Chord maintains its routing information as nodes join and leave the system, with high probability each such event results in no more than $O(\log^2 N)$ messages, performance degrades gracefully when that information is out of date, only one piece information per node need be correct in order for Chord to guarantee correct routing of queries

13.3 Related Work

Chord vs. DNS: pp.1 RHS, Chord requires no special servers vs. DNS relies on a set of special root servers, Chord imposes no naming structure vs. DNS names are structured to reflect administrative boundaries, Chord can also be used to find data objects that are not tied to particular machines vs. DNS is specialized to the task of finding named hosts or services

Chord vs. Freenet: pp.2 LHS, Chord does not provide anonymity but its lookup operation runs in predictable time and always results in success or definitive failure vs. Freenet's lookups take the form of searches for cached copies which allows it to provide a degree of anonymity but prevents it from guaranteeing retrieval of existing documents or from providing low bounds on retrieval costs

Chord vs. Globe: pp.2 LHS, Chord performs hash function well enough that it can achieve scalability without also involving any hierarchy although Chord does not exploit network locality as well as Globe vs. Globe arranges the internet as a hierarchy of geographical, topological, or administrative domains effectively constructing a static world-wide search tree much like DNS

Chord vs. Plaxton: pp.2 LHS, Chord is substantially less complicated and handles concurrent node joins and failures well vs. Plaxton guarantees that queries make a logarithmic number hops and that keys are well balanced like Chord, but it also ensures that queries never travel further in network distance than the node where the key is stored subject to assumptions about network topology

Chord vs. CAN: pp.2 LHS, Chord's routine procedure may be thought of as a one-dimensional analog of the grid location system vs. CAN uses a d-dimensional Cartesian coordinate space for some fixed d to implement a distributed hash table that maps keys onto values, each node maintains $O(d)$ state and the lookup cost is $O(dN^{1/d})$ thus the state maintained by a CAN node does not depend on the network size N but the lookup cost increases faster than $\log N$, only when $d = \log N$ do CAN lookup times and storage needs match Chord's

pp.2 LHS, Chord can help avoid single points of failure or control and the lack of scalability

13.4 System Model

pp.2 RHS, Chord addresses the following difficult problems

load balance: Chord acts as a distributed hash function spreading keys evenly over the nodes

decentralization: Chord is fully distributed, no node is more important than any other, this improves robustness

scalability: the cost of a Chord lookup grows as the log of the number of nodes, no parameter tuning is required to achieve this scaling

availability: Chord automatically adjusts its internal tables to reflect newly joined nodes as well as node failures, this is true even if the system is in a continuous state of change

flexible naming: Chord places no constraints on the structure of the keys it looks up—the Chord keyspace is flat

pp.2 RHS, the Chord software takes the form of a library to be linked with the client and server applications that use it, the application interacts with Chord in two main ways—(1) Chord provides a lookup(key) algorithm that yields the IP addresses of the node responsible for the key (2) the Chord software on each node notifies the application of changes in the set of keys that the node is responsible for
pp.3 LHS, examples of applications of Chord include large-scale combinatorial search such as code breaking

13.5 The Base Chord Protocol

pp.3 LHS, this section describes a simplified version of the protocol that does not handle concurrent joins or failures

13.5.1 Overview

pp.3 LHS, Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them, with high probability the hash function balances load, also with high probability when an Nth node joins or leaves the network only an $O(1/N)$ fraction of the keys are moved to a different location, Chord improves the scalability of consistent hashing by avoiding the requirement that every node know about every other node, in an N-node network each node maintains information only about $O(\log N)$ other nodes and a lookup requires $O(\log N)$ messages, Chord must update the routing information when a node joins or leaves the network, a join or leave requires $O(\log^2 N)$ messages

13.5.2 Consistent hashing

pp.3 RHS, the consistent hash function assigns each node and key an m-bit identifier using a base hash function such as SHA-1, a node's identifier is chosen by hashing the node's IP address while a key identifier is produced by hashing the key, we will use the term "key" and "node" to refer to both the key and node and their identifiers, the identifier length m must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible
pp.3 RHS, consistent hashing assigns keys to nodes as follows: identifiers are ordered in an *identifier circle* modulo 2^m, key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier space, this node is called the *successor node* of key k denoted by $successor(k)$, if identifiers are represented as a circle of numbers from 0 to $2^m - 1$ then $successor(k)$ is the first node clockwise from k, to maintain the consistent hashing mapping when a node n joins the network, certain keys previously assigned to n's successor now become assigned to n, when node n leaves the network all of its assigned keys are reassigned to n's successor, no other changes in assignment of keys to nodes need occur, when consistent hashing is implemented as described above we have

THEOREM 1: for any set of N nodes and K keys, with high probability

- each node is responsible for at most $(1 + \epsilon)K/N$ keys where $\epsilon = O(\log N)$
- when an $(N + 1)$st node joins or leaves the network responsibility for $O(K/N)$ keys changes hands

pp.4 LHS, rather than using a k-universal hash function we chose to use the standard SHA-1 function as our base hash function, this makes our protocol deterministic, thus instead of stating that our theorems hold with high probability we can claim that they hold "based on standard hardness assumptions"

13.5.3 Scalable key location

pp.4 LHS, each node need only be aware of its successor node on the circle, queries for a given identifier can be passed around the circle via these successor pointers until they first encounter a node that succeeds the identifier, this is the node the query maps to, to accelerate this process Chord maintains additional routing information, this additional information is not essential for correctness which is achieved as long as the successor information is maintained correctly, each node n maintains a routing table with (at most) m entries called the *finger table*, the ith entry in the table at node n contains the identity of the first node s that succeeds n by at least 2^{i-1} on the identifier circle i.e. $s = successor(n + 2^{i-1})$ where $1 \leq i \leq m$, we call node s the ith *finger* of node n and denote it by $n.finger[i].node$, a finger table entry includes both the Chord identifier and the IP address and port number of the relevant node, node that the first finger of n is its successor $s = successor(n)$
pp.4 RHS, the definition of variables for node n are

- $finger[i].start = (n + 2^{i-1}) \bmod 2^m$ where $1 \leq i \leq m$
- $finger[i].node =$ first node $\geq n.finger[i].start$
- $finger[i].interval = [finger[i].start, finger[i+1].start)$

pp.4 RHS, when a node n does not know the successor of a key k, n searches its finger table for the node j whose ID most immediately precedes k, and asks j for the node it knows whose ID is closest to k, by repeating this process n learns about nodes with IDs closer and closer to k, the finger pointers at repeatedly doubling distances around the circle cause each iteration of the loop in find_predecessor to halve the distance to the target identifier, from this intuition follows a theorem

> THEOREM 2: with high probability or under standard hardness assumptions, the number of nodes that must be contacted to find a successor in an N-node network is $O(\log N)$

13.5.4 Node joins

pp.5 LHS, to preserve the ability to locate every key in the network when a node joins or leaves the network, Chord needs to preserve two invariants—(1) each node's successor is correctly maintained (2) for every key k node $successor(k)$ that is responsible for k, to simplify the join and leave mechanism each node in Chord maintains a *predecessor pointer*, a node's predecessor pointer contains the Chord identifier and IP address of the immediate predecessor of that node, and can be used to walk counter-clockwise around the identifier circle
pp.6 LHS, we assume that a new node learns the identity of an existing node by some external mechanism, to preserve the invariants stated above Chord must perform three tasks when a node n joins the network

initializing fingers and predecessor: node n learns its predecessor and fingers by asking n' to look them up, naively performing find_successor for each of the m finger entries would give a runtime of $O(m \log N)$, to reduce this n checks whether the ith finger is also the correct $(i+1)$th finger for each i, this happens when $finger[i].interval$ does not contain any node and thus $finger[i].node \geq finger[i+1].start$, it can be shown that the change reduces the expected number of finger entries that must be looked up to $O(\log N)$ which reduces the overall time to $O(\log^2 N)$

updating fingers of existing nodes: node n will need to be entered into the finger tables of some existing nodes, node n will become the ith finger of node p if and only if (1) p precedes n by at least 2^{i-1} and (2) the ith finger of node p succeeds n, the first p that can meet these two conditions is the immediate predecessor of $n - 2^{i-1}$, thus for a given n the algorithm starts with the ith finger of node n and then continues to walk in the counterclockwise direction on the identifier circle until it encounters a node whose ith finger precedes n

transferring keys: node n can become the successor only for keys that were previously the responsibility of the node immediately following n, so n only needs to contact that one node to transfer responsibility for all relevant keys

pp.5 RHS, its performance can be summarized by the following theorem

> THEOREM 3: with high probability any node joining or leaving an N-node Chord network will use $O(\log^2 N)$ messages to re-establish the Chord routing invariants and finger tables

13.6 Concurrent Operations and Failures

13.6.1 Stabilization

pp.7 LHS, a basic stabilization protocol is used to keep nodes' successor pointers up to date which is sufficient to guarantee correctness of lookups, those successor pointers are then used to verify and correct finger table entries, our stabilization scheme guarantees to add nodes to a Chord ring in a way that preserves reachability of existing nodes even in the face of concurrent joins and lost and reordered messages, every node runs stabilize periodically (this is how newly joined nodes are noticed by the network), when node n runs stabilize it asks n's successor for the successor's predecessor p and decides whether p should be n's successor instead, this would be the case if node p recently joined the system, stabilize also notifies node n's successor of n's existence giving the successor the chance to change its predecessor to n, the successor does this only if it knows of no closer predecessor than n;
pp.7 RHS, newly joined nodes that have not yet been fingered may cause find_predecessor to initially undershoot, but the loop in the lookup algorithm will nevertheless follow successor pointers through the newly joined nodes until the correct predecessor is reached, eventually fix_fingers will adjust finger table entries eliminating the need for these linear scans
pp.7 RHS, the following theorems show that all problems caused by concurrent joins are transient, the theorems assume that any two nodes trying to communicate will eventually succeed

> THEOREM 4: once a node can successfully resolve a given query, it will always be able to do so in the future
> THEOREM 5: at some time after the last join all successor pointers will be correct

pp.7 RHS, new joins influence the lookup only by getting in between the old predecessor and successor of a target query, these new nodes may need to be scanned linearly, but unless a tremendous number of nodes join the system, the number of nodes between two old nodes is likely to be very small, so the impact on lookup is negligible, formally we can state the following theorem

> THEOREM 6: if we take a stable network with N nodes and another set of up to N nodes join the network with no finger pointers but correct successor pointers, then lookups will still take $O(\log N)$ time with high probability

more generally so long as the time it takes to adjust fingers is less than the time it takes the network to double in size, lookups should continue to taken $O(\log N)$ hops

13.6.2 Failures and replication

pp.8 LHS, the key step in failure recovery is maintaining correct successor pointers since in the worst case find_predecessor can make progress using only successors, to help achieve this each Chord node maintains a "successor-list" of its r nearest successors on the Chord ring, in ordinary operation a modified version of the stabilize routine maintains the successor-list, if node n notices that its successor has failed it replaces it with the first live entry in its successor-list, at that point n can direct ordinary lookups for keys for which the failed node was the successor to the new successor

pp.8 LHS, after a node failure but before stabilization has completed, other nodes may attempt to send requests through the failed node as part of a find_successor lookup, ideally the lookups would be able to proceed after a timeout by another path despite the failure, all that is needed is a list of alternate nodes easily found in the finger table entries preceding that of the failed node
pp.8 LHS, the technical report proves the following two theorem that show that the successor-list allows lookups to succeed and be efficient even during stabilization

> THEOREM 7: if we use a successor list of length $r = O(\log N)$ in a network that is initially stable and every node fails with probability 1/2, then with high probability find_successor returns the closest living successor to the query key
> THEOREM 8: if we use a successor list of length $r = O(\log N)$ in a network that is initially stable and every node fails with probability 1/2, then the expected time to execute find_successor in the failed network is $O(\log N)$

pp.8 LHS, the successor-list mechanism also helps higher layer software replicate data, the fact that a Chord node keeps track of its r successors means that it can inform the higher layer software when successors come and go and thus when the software should propagate new replicas

13.7 Simulation and Experimental Results

13.7.1 Protocol simulator

pp.8 RHS, the Chord protocol can be implemented in an iterative or recursive style

iterative: a node resolving a lookup initiates all communication, it asks a series of nodes for information from their finger tables

recursive: each intermediate node forwards a request to the next node until it reaches the successor

the simulator implements the protocols in an iterative style

13.7.2 Load balance

pp.8 RHS, node identifiers do not uniformly cover the entire identifier space, the probability that a particular bin does not contain any node is $(1-1/N)^N$, for large values of N this approaches $e^{-1} = 0.368$, the consistent hashing paper solves this problem by associating keys with virtual nodes and mapping multiple virtual nodes with unrelated identifiers to each real node; pp.9 LHS, adding virtual nodes as an indirection layer can significantly improve load balance, the tradeoff is that routing table space usage will increase as each actual node now needs r times as much space to store the finger tables for its virtual nodes

13.7.3 Path length

pp.9 LHS, in the context of Chord we define the path length as the number of nodes traversed during a lookup operation, the mean path length increases logarithmically with the number of nodes which is about $\log N/2$, the reason for the 1/2 is that in general the number of fingers we need to follow will be the number of 1s in the binary representation of the distance from node to query, since the distance is random we expect half the $\log N$ bits to be 1s

13.7.4 Lookups during stabilization

pp.10 LHS, a lookup issued after some failures but before stabilization has completed may fail for two reasons—(1) the node responsible for the key may have failed (2) some nodes' finger tables and predecessor pointers may be inconsistent due to concurrent joins and node failures

13.8 References

consistent hashing: KARGER, D., LEHMAN, E., LEIGHTON, F., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: *Distributed caching protocols for relieving hot spots on the World Wide Web*, in Proceedings of the 29th Annual ACM Symposium on Theory of Computing (El Paso, TX, May 1997), pp.654—663.

technical report detailing the proofs: STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: *A scalable peer-to-peer lookup service for internet applications*, Tech. Rep. TR-819, MIT LCS, March 2001, `http://www.pdos.lcs.mit.edu/chord/papers/`

14 Lesson #11 Designing Distributed Algorithms for Mobile Computing Networks

14.1 The Publication Details

Author: B. R. Badrinath, Arup Acharya, Tomasz Imielinski

Title: Designing Distributed Algorithms for Mobile Computing Networks

Journal: Computer Communications, Vol. 19, Issue 4, April 1996, pp.309–320

DOI: `https://web.mst.edu/~madrias/cs401-02/struc-1.pdf`

14.2 Abstract

pp.1, to overcome the resource constraints of mobile hosts we propose two tier principle for structuring distributed algorithms for mobile hosts, in addition since location of a mobile host could change after initiating a distributed computation and before receiving the result location management of mobile participants need to be explicitly integrated with algorithm design

14.3 Introduction

pp.2 LHS, the bandwidth of the wireless link connecting a MH to a MSS is significantly lower than the wired links between static hosts, in addition MHs have tight constraints on power consumption and transmission and reception of messages over the wireless link also consumes power at a MH, therefore distributed algorithms need to minimize communication over the wireless links, we thus propose a two tier principle for structuring distributed algorithms

- the computation and communication costs of an algorithm are borne by the static portion of the network, the core objective of the algorithm is achieved through a distributed execution amongst the fixed hosts
- performing only those operations at the mobile hosts that are necessary for the desired overall functionality

in conjunction with this principle to deliver the desired result to a migrant MH a distributed algorithm must now explicitly incorporate location management of migrant MHs in its design

14.4 The System Model

pp.2 RHS, a host that can move while retaining its network connections is a mobile host (MH), the infrastructure machines that communicate directly with the mobile hosts are called mobile support stations (MSS), a "cell" is a logical geographical coverage area under a MSS, all MHs that have identified themselves with a particular MSS are considered to be local to the MSS, a MH can directly communicate with a MSS and vice versa only if the MH is physically located within the cell services by the MSS, at any given instant of time a MH may belong to only one cell, its current cell defines a MH's "location", host mobility is represented in this model as migration of MHs between cells, all fixed hosts and the communication paths between them constitute the static/fixed network; pp.3 LHS, the static network provides reliable sequenced delivery of messages between any two MSSs with arbitrary message latency, the wireless network within a cell ensures FIFO delivery of messages between a MSS and a local MH

join/leave: a MH is required to send a leave(r) message on the MH-to-MSS channel supplying the sequence number r of the last message received on the MSS-to-MH channel, after sending this message the MH neither sends nor receives any other message within the current cell, each MSS maintains a list of IDs of MHs that are local to its cell, on receipt of leave() from a local MH it is deleted from the list, conversely when a MH enters a new cell it sends a join($mh - id$) to the MSS, it is then added to the list of local MHs at the new MSS

disconnect/reconnect: a MH disconnects by sending a disconnect(r) message to its local MSS M where r is the sequence number of the last message it received from M similar to leave(r) message, M deletes the MH from its list of local MHs, however it sets a "disconnected" flag for the particular $mh - id$, when some other MSS M' attempts to contact this MH after it is disconnected from M, M informs M' of the disconnected status of the MH, later the MH may reconnect at a MSS N by sending a reconnect($mh - id, M$) message, N informs M of the MH's reconnection and as a result M unsets the "disconnected" flag for the MH while N adds it to its list of local MHs ⇒ a MH can disconnect without leave to save power

pp.3 LHS, message communication from a MH h_1 to another MH h_2 occurs as follows: h_1 first sends the message to its local MSS M_1 using the wireless link, M_1 forwards it to M_2—the local MSS of h_2 via the fixed network, M_2 then transmits it to h_2 over its local wireless network
pp.2 RHS & pp.3 LHS, in this paper we assume

- all hosts and communication links are reliable
- all fixed hosts act as MSSs (and use the terms MSS and fixed host interchangeably)
- the number of MSSs denoted by N_{mss} and that of MH denoted by N_{mh} satisfy $N_{\text{mh}} \gg N_{\text{mss}}$
- any message destined for a mobile host incurs a fixed search cost

pp.3 RHS, our system model contains three cost measures for counting the number of messages exchanged

- C_{fixed} = cost of sending a point-to-point message between any two fixed hosts
- C_{wireless} = cost of sending a message from a MH to its local MSS over the wireless channel and vice versa
- C_{search} = cost incurred to locate a MH and forward a message to its current local MSS from a source MSS

based on the above cost parameters a message sent from an MH to another MH incurs a cost of $2C_{\text{wireless}} + C_{\text{search}}$ while a message sent from a MSS to a non-local MH incurs a cost of $C_{\text{search}} + C_{\text{wireless}}$

14.5 Structuring Distributed Algorithms

14.5.1 Motivation

pp.3 RHS, we cast distributed systems with mobile hosts into a two-tier structure

- a network of fixed hosts with more resources in terms of storage, computing, and communication
- mobile hosts which may operate in a disconnected or doze mode, connected by a low-bandwidth wireless connection to this network

pp.4 LHS, communication necessary to execute an algorithm may be split into three components

global: messages whose source and destination are both fixed hosts, mostly representing the communication necessary for the progress of an algorithm execution

local: communication within a single wireless cell between a MH and its local MSS, used to initiate an algorithm execution from a MH or to communicate the final result of an execution to a MH

search: messages that the fixed hosts exchange to determine the current location of a MH

our approach suggests that the global component dominate the overall communication
pp.4 LHS, the two unique modes of operation of MHs are disconnected and doze-mode

doze-mode: the MH shuts/slows down most of its system functions to reduce power consumption and only listens for incoming messages, but remains reachable from the rest of the system and can be induced by the system to resume its normal operating mode

disconnected: cut off from the system in the intervening period

a distributed algorithm designed for the mobile computing environment should not require each MH to participate in every execution of the algorithm, otherwise it prevents those MHs from operating in a doze-mode and has to handle a variable number of participants and incur a search overhead if some MH disconnects while an execution is in progress; pp.4 RHS, the two-tier principle makes it easy to handle disconnections—since the fixed hosts are responsible for the progress to an algorithm execution disconnection of one or more MHs does not alter the number of participants in the algorithm, note that since a MH can download any data to the fixed network that is necessary for progress of the algorithm and inform the system prior to an impending disconnection, disconnection should not be associated with the same semantics as failure; pp.7 LHS, disconnection of a MH that does not need to access the token has no effect on the algorithm execution

14.5.2 Structuring a token-based logical ring for mobile hosts: a case study

pp.5 LHS, a fundamental algorithm in distributed systems consisted of circulating a token amongst participants in a logical ring, each participant executes as follows

- wait receipt of token from its predecessor in the ring
- enter critical region if desired
- send token to its successor in the ring

the algorithm trivially satisfies two important properties

- mutual exclusion is trivially guaranteed to the current holder of the token
- it allows fair access to the token by allowing each participant to access the token at most once in one traversal of the ring

14.5.3 Restructuring the logical ring using the two-tier principle

pp.5 RHS, the two-tier principle suggests that the logical ring should be established within the fixed network, the logical ring now consists of all MSSs with the token visiting each MSS in a predefined sequence, a MH that wishes to access the token is required to submit a request to its local MSS, when the token visits this MSS all pending requests are serially serviced; pp.6 LHS, an alternative method of structuring the ring is to partition the set of all MSSs into "areas" and associate a designated fixed host called a "proxy" with each area, the token now circulates only amongst the proxies and each proxy is responsible for servicing token requests from MSSs within its area

14.5.4 SEARCH strategy

14.5.4.1 actions executed by a MSS

- on receipt of a request for the token from a local MH h, M adds the request to the rear of its *request queue*
- on receipt of the token from its predecessor in the ring M executes the following steps
  1. entries from the *request queue* are moved to the *grant queue*
  2. loop through the following until the *grant queue* is empty
     - remove the request at the head of the *grant queue*
     - if h is currently local to M then deliver the token to h over the local wireless link
     - else search and forward the token to the MSS h is currently local to
     - await return of the token from h
  3. forward token to M's successor in the logical ring

14.5.4.2 actions executed by a MH

- when h needs access to the token, it submits a request to its current local MSS M
- when h receives the token from M (may not be currently local) it accesses the critical region and then returns the token to the same MSS

the above algorithm assumes that a MH

- does not submit a second request if its previous request has not yet been serviced
- may not disconnect permanently after receiving the token

14.5.4.3 correctness sketch

pp.6 LHS, starvation does not occur i.e. every request submitted is eventually granted, because the token cannot reside forever at a fixed host and thus it eventually visits every fixed host in the ring, since the maximum number of requests serviced by a MSS holding the token is bounded—only those requests that were made prior to arrival of the token are serviced, it follows that the *grant queue* contains at most N_{mh} requests one per MH

14.5.4.4 communication cost

- cost incurred by the token for one traversal of the logical ring: $N_{\text{mss}} \times C_{\text{fixed}}$
- cost of submitting a request from a MH to its local MSS: C_{wireless}
- cost of delivering a token (for a migrant MH, worst case): $C_{\text{search}} + C_{\text{wireless}}$
- cost of returning the token to the sender MSS (for a migrant MH, worst case): $C_{\text{fixed}} + C_{\text{wireless}}$

thus the (worst case) cost of processing a request is $3C_{\text{wireless}} + C_{\text{fixed}} + C_{\text{search}}$, and thus the total cost of processing K requests in one traversal of the ring is $K \times (3C_{\text{wireless}} + C_{\text{fixed}} + C_{\text{search}}) + N_{\text{mss}} \times C_{\text{fixed}}$

14.5.5 INFORM strategy

pp.7 LHS, this strategy requires the MH to notify the MSS where it submitted its request after every change in its location till it receives the token

14.5.5.1 actions executed by a MSS

- on receipt of a request for the token from a local MH h, M adds a request $\langle h, M\rangle$ to the rear of its *request queue*
- upon receipt of a $\mathsf{inform}(h, M')$ message the current value of $loc(h)$ is replaced with M' in the entry $\langle h, loc(h)\rangle$ in M's *request queue*
- on receipt of the token from its predecessor in the ring M executes the following steps
  1. entries from the *request queue* are moved to the *grant queue*
  2. loop through the following until the *grant queue* is empty
     - remove the request $\langle h, loc(h)\rangle$ at the head of the *grant queue*
     - if $loc(h) = M$ then deliver the token to h over the local wireless link
     - else forward the token to $loc(h)$ the MSS currently local to h which will transmit it to h
     - await return of the token from h
  3. forward token to M's successor in the logical ring

14.5.5.2 actions executed by a MH

- when h needs access to the token, it submits a request to its current local MSS M and stores M in the local variable *req_loc*
- when h receives the token from the MSS *req_loc* it accesses the critical region, returns the token to the same MSS, and then sets *req_loc* to $\perp$
- after every move h sends $\mathsf{join}(h, req\_loc)$ message to the MSS M' upon entering the cell under M', if *req_loc* received with the $\mathsf{join}()$ message is not $\perp$ then M' sends an $\mathsf{inform}(h, M')$ message to the MSS *req_loc*

14.5.5.3 comparison of SEARCH and INFORM strategies

pp.7 RHS, assume h makes MOB number of moves in the intervening period, the inform cost of the INFORM strategy is $MOB \times C_{\text{fixed}}$, on the other hand in the SEARCH strategy M would search for the current location of h and the cost incurred would be C_{search}, thus the INFORM strategy is preferable to the SEARCH strategy when $MOB \times C_{\text{fixed}} < C_{\text{search}}$

14.5.6 PROXY strategy

pp.7 RHS, while the SEARCH strategy is useful for migrant MHs that frequently change their cells, the INFORM strategy is better for migrant MHs that change their locations less often, the PROXY strategy combines advantages of both SEARCH and INFORM strategies, and is tuned for a mobility pattern wherein a migrant MH moves frequently between adjacent cells while rarely moving between non-adjacent cells—the set of all MSSs is partitioned into "areas" and MSSs within the same area are associated with a common proxy, a MH makes a "wide area" move when its local MSSs before and after the move are in different areas, analogously a "local area" move occurs when its proxy does not change after a move

14.5.6.1 actions executed by a proxy

- on receipt of a request for the token from a MH h forwarded by a MSS within P's local area, P adds a request $\langle h, P\rangle$ to the rear of its *request queue*
- upon receipt of a $\mathsf{inform}(h, P')$ message the current value of $proxy(h)$ is replaced with P' in the entry $\langle h, proxy(h)\rangle$ in P's *request queue*
- on receipt of the token from its predecessor in the ring P executes the following steps
  1. entries from the *request queue* are moved to the *grant queue*
  2. loop through the following until the *grant queue* is empty
     - remove the request $\langle h, proxy(h)\rangle$ at the head of the *grant queue*
     - if $proxy(h) = P$ then deliver the token to h after searching for h within the MSSs under P
     - else forward the token to $proxy(h)$, the proxy currently local to h, which will transmit it to h after a local search within its area
     - await return of the token from h
  3. forward token to P's successor in the logical ring

14.5.6.2 actions executed by a MH

- when h needs access to the token, it submits a request to its current local MSS M under P and stores P in the local variable *init_proxy*
- when h receives the token from *init_proxy* it accesses the critical region, returns the token to *init_proxy*, and then sets *init_proxy* to $\perp$
- after every move h sends $\mathsf{join}(h, init\_proxy)$ message to the MSS M' under P' upon entering the cell under M', if *init_proxy* received with the $\mathsf{join}()$ message is not $\perp$ and is different from P' then M' sends an $\mathsf{inform}(h, P')$ message to *init_proxy*

14.5.6.3 communication cost

pp.8 LHS, prior to delivering the toaken to a MH a proxy needs to locate a MH amongst the MSSs within its area, we refer to this as a *local search* with an associated cost $C_{\text{l-search}}$; pp.8 RHS, if $N_{\text{area}} = N_{\text{mss}}/N_{\text{proxy}}$ is the number of MSSs within the area then $C_{\text{l-search}} = (N_{\text{mss}}/N_{\text{proxy}} + 1) \times C_{\text{fixed}}$

- cost incurred by the token for one traversal of the logical ring: $N_{\text{proxy}} \times C_{\text{fixed}}$
- cost of submitting a request from a MH to its local proxy: $C_{\text{wireless}} + C_{\text{fixed}}$

- cost of delivering a token (for a migrant MH, worst case): $C_{\text{l-search}} + C_{\text{wireless}} + C_{\text{fixed}}$
- cost of returning the token to the sender proxy (for a migrant MH, worst case): $C_{\text{fixed}} + C_{\text{wireless}}$

thus the (worst case) cost of processing a request is $3C_{\text{wireless}} + 3C_{\text{fixed}} + C_{\text{l-search}}$, and thus the total cost of processing K requests in one traversal of the ring is $K \times (3C_{\text{wireless}} + C_{\text{fixed}} + C_{\text{search}}) + MOB_{\text{wide}} \times C_{\text{fixed}}$ where MOB_{wide} is the number of wide-area moves

14.5.7 Ensuring fair access to the token regardless of mobility

pp.9 RHS, below we present a scheme that is applicable to all three location management strategies which prevents a MH from accessing the token more than once in one traversal of the ring

1. the token is associated with a loop counter (*token_val*) which is incremented every time it completes a traversal
2. each MH maintains a local counter *access_count* whose current value is sent along with the MH's request for the token
3. a pending request is moved from the *request queue* to the *grant queue* at the MSS or proxy holding the token only if the request's *access_count* is less than the token's current *token_val*
4. when a MH receives the token it assigns the current value of *token_val* to its copy of *access_count*

this scheme represents a trade-off between fairness of token access amongst the MHs and satisfying as many token requests as possible per traversal

14.6 Handling Mobility by Replicating Token Requests

pp.9 RHS, the advantage of using a replication-based approach is that the location of migrant MHs does not need to be explicitly managed, since regardless of its current location the local MSS has a copy of a migrant MH's pending request, however mutual exclusion requires that requests from MHs be globally ordered amongst all MSSs and only the first request in this ordered sequence be serviced at any time, the choice of the servicing MSS is determined by the current location of the MH that made this request—its local MSS can unilaterally decide to service the request with the assurance that no other MSS will simultaneously process any another request

14.6.0.1 actions executed by a MH

- each MH h maintains a local counter *h_count* which is incremented prior to submitting a new request for mutual exclusion, then a request for mutual exclusion $\mathsf{req}(h, h\_count)$ is submitted to its local MSS
- on a move h includes its current value of *h_count* with the $\mathsf{join}()$ message to its local MSS
- when a MH receives a $\mathsf{grant}()$ message from its local MSS, it accesses the shared resource and then replies with a $\mathsf{release}(h, h\_count)$ message to its current local MSS

14.6.0.2 actions executed by a MSS

- each MSS maintains a *delivery queue* of pending requests, each request is flagged as either deliverable or undeliverable and is assigned a priority number, the queue is kept sorted in increasing order of message priority number
- on receipt of the request req(h, h_count) from a MH h, M executes a two-phase protocol to order this request relative to all other pending requests

  phase I: M sends a copy of req(h, h_count) to all other MSSs, each receiving MSS assigns a temporary priority number to the received request from M that is greater than any of the requests currently in its *delivery queue*, the request is tagged undeliverable and inserted at the end of the queue, the temporary priority number is sent back to the initiator M

  phase II: M computes the maximum of all temporary priority numbers and sends it to all MSSs, each MSS then assigns this number as the final priority number of the request and tags it deliverable and re-sorts its *delivery queue*
- a MSS can service the pending request if (1) the request is tagged deliverable (2) the MH h is local to its cell (3) the h_count value submitted by h either with the join() message or the req() message is equal to *count*, i.e. the request has not already been serviced by another MSS, if these conditions are satisfied then the MSS sends a grant() message to h
- on receiving a release(h, h_count) message from a local MH h, a MSS first deletes the entry req(h, h_count) from its *delivery queue*, it then sends a delete(h, h_count) message to all other MSSs which then delete the corresponding entry from their respective *delivery queue*

pp.10 RHS, correctness of the message-ordering portion of the above algorithm follows directly from the correctness of the two-phase ABCAST protocol, however unlike ABCAST the protocol we need an explicit delete() message to ensure mutual exclusion, we also assign a counter value with each request to avoid a MH receiving grant() messages from more than one MSS for the same request

14.6.0.3 communication costs

pp.10 RHS, each execution of the above algorithm incurs a total cost of $(4N_{\text{mss}} - 1) \times C_{\text{fixed}} + 3C_{\text{wireless}}$, note that it involves no search component i.e. the cost is independent of a MH's mobility

14.7 Conclusions

pp.11 LHS, this paper first presents a new system model for the mobile computing environment and then describes a general principle for structuring distributed algorithms in this model, by performing only those operations at the mobile hosts that are necessary for the desired overall functionality power consumption at the mobile hosts is kept to a minimum, since updates to the data structure are performed at the fixed hosts the overall search cost is reduced

pp.11 RHS, the two tier principle by itself is not sufficient to handle the effects of varying location of MHs, this required location management of migrant MHs and we presented three strategies: (1) search (2) inform (3) search within a local area with location updates after wide-area moves

pp.11 RHS, replicating the queue of pending requests at all MSSs eliminated the need for location management strategies for migrant MHs, but increased the number of messages exchanged within the fixed network to globally order pending requests among all MSSs

14.8 References

two-phase ABCAST protocol: K. Birman and T. Joseph, *Reliable communications in presence of failures*, ACM Trans. Comput. Systems, 5 (1) (1987) pp.47–65

15 Lesson #12 A Fault-Tolerant Abstraction for In-Memory Cluster Computing

15.1 The Publication Details

Author: Matei Zaharia et al.

Title: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Journal: NSDI 12, 9th USENIX Symposium on Network Systems Design and Implementation

DOI: `https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf`

15.2 Abstract

pp.1 LHS, resilient distributed datasets or RDDs are motivated by two types of applications that current computing frameworks handle inefficiently—iterative algorithms and interactive data mining tools

15.3 Introduction

pp.1 LHS, although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory; pp.1 RHS, the main challenge in designing RDDs is defining a programming interface that can provide fault tolerance efficiently, RDDs provide an interface based on coarse-grained transformations that apply the same operation to many data items, this allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data; pp.2 LHS, RDDs are a good fit for many parallel applications because these applications naturally apply the same operation to multiple data items

15.4 Resilient Distributed Datasets (RDDs)

15.4.1 RDD abstraction

pp.2 LHS, an RDD is a read-only partitioned collection of records, RDDs can only be created through deterministic operations on either data in stable storage or other RDDs; pp.2 RHS, RDDs do not need to be materialized at all times, instead an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage, finally users can control two other aspects of RDDs

persistence: users can indicate which RDDs they will reuse and choose a storage strategy for them (e.g. in-memory storage)

partitioning: users can ask that an RDD's elements be partitioned across machines based on a key in each record, this is useful for placement optimizations such as ensuring that two datasets that will be joined together are hash-partitioned in the same way

15.4.2 Spark programming interface

pp.2 RHS, programmers start by defining one or more RDDs through *transformations* (e.g. map, filter, and join) on data in stable storage, they can then use these RDDs in *actions* which are operations that return a value to the application or export data to a storage system (e.g. count, collect, save), Spark

computes RDDs lazily the first time they are used in an action so that it can pipeline transformations, in addition programmers can call a persist method to indicate which RDDs they want to reuse in future operations, Spark keeps persistent RDDs in memory by default but it can spill them to disk if there is not enough RAM, users can also request other persistence strategies such as storing the RDD only on disk or replicating it across machines through flags to persist, finally users can set a persistence priority on each RDD to specify which in-memory data should spill to disk first

15.4.3 Advantages of the RDD model

- pp.3 RHS, the main difference between RDDs and DSM is that RDDs can only be created through coarse-grained transformations while DSM allows reads and writes to each memory location, this restricts RDDs to applications that perform bulk writes but allows for more efficient fault tolerance, in particular RDDs do not need to incur the overhead of checkpointing, furthermore only the lost partitions of an RDD need to be recomputed upon failure and they can be recomputed in parallel on different nodes

- a second benefit of RDDs is that their immutable nature lets a system mitigate slow nodes (stragglers) by running backup copies of slow tasks as in MapReduce, backup tasks would be hard to implement with DSM as the two copies of a task would access the same memory locations and interface with each other's updates

- finally RDDs provide two other benefits over DSM, first in bulk operations on RDDs a runtime can schedule tasks based on data locality to improve performance, second RDDs degrade gracefully when there is not enough memory to store them as long as they are only being used in scan-based operations

15.4.4 Applications not suitable for RDDs

pp.4 LHS, RDDs are best suited for batch applications that apply the same operation to all elements of a dataset, RDDs would be less suitable for applications that make asynchronous fine-grained updates to shared state such as a storage system for a web application or an incremental web crawler, for these applications it is more efficient to use systems that perform traditional update logging and data checkpointing such as databases, our goal is to provide an efficient programming model for batch analytics and leave these asynchronous applications to specialized systems

15.5 Spark Programming Interface

pp.4 LHS, Spark provides the RDD abstraction through a language-integrated API, to use Spark developers write a *driver* program that connects to a cluster of *workers*, the driver defines one or more RDDs and invokes actions on them, Spark code on the driver also tracks the RDDs' lineage, the workers are long-lived processes that can store RDD partitions in RAM across operations

15.5.1 RDD operations in Spark

pp.4 RHS, users can get an RDD's partition order which is represented by a Partitioner class

15.6 Representing RDDs

pp.6 LHS, we propose representing each RDD through a common interface that exposes five pieces of information—a set of *partitions* which are atomic pieces of the dataset, a set of *dependencies* on parent RDDs, a function for computing the dataset based on its parents, and metadata about its partitioning

scheme and data placement
pp.6 RHS, we found it both sufficient and useful to classify dependencies into two types

narrow dependencies: where each partition of the parent RDD is used by at most one partition of the child RDD, e.g. map

wide dependencies: where multiple child partitions may depend on it, e.g. join

this distinction is useful for two reasons

- narrow dependencies allow for pipelined execution on one cluster node which can compute all the parent partitions vs. wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation
- recovery after a node failure is more efficient with a narrow dependency as only the lost parent partitions need to be recomputed and they can be recomputed in parallel on different nodes vs. in a lineage graph with wide dependencies a single failed node might cause the loss of some partition from all the ancestors of an RDD requiring a complete re-execution

pp.6 RHS, this common interface for RDDs made it possible to implement most transformations in Spark, e.g.

HDFS files: partitions returns one partition for each block of the file, preferredLocations gives the nodes the block is on, iterator reads the block

map: calling map on any RDD returns a MappedRDD object which has the same partitions and preferred locations as its parent

union: calling union on two RDDs returns an RDD whose partitions are the union of those of the parents, each child partition is computed through a narrow dependency on the corresponding parent

join: joining two RDDs may lead to either two narrow dependencies (if they are both hash/range partitioned with the same partitioner), two wide dependencies, or a mix (if one parent has a partitioner and one does not)

15.7 Implementation

pp.7 LHS, the system runs over the Mesos cluster manager, Spark can read data from any Hadoop input source (e.g. HDFS or HBase) using Hadoop's existing input plugin APIs

15.7.1 Job scheduling

pp.7 RHS, overall our scheduler is similar to Dryad's, but it additionally takes into account which partitions of persistent RDDs are available in memory, whenever a user runs an action on an RDD the scheduler examines that RDD's lineage graph to build a DAG of stages to execute, each stage contains as many pipelined transformations with narrow dependencies as possible, the boundaries of the stages are the shuffle operations required for wide dependencies or any already computed partitions that can short-circuit the computation of a parent RDD, our scheduler assigns tasks to machines based on data locality using delay scheduling, for wide dependencies we currently materialize intermediate records on the nodes holding parent partitions to simplify fault recovery much like MapReduce materializes map outputs

15.7.2 Interpreter integration

pp.8 LHS, we made two changes to the interpreter in Spark

class shipping: to let the worker nodes fetch the bytecode for the classes created on each line, we made the interpreter serve these classes over HTTP

modified code generation: we modified the code generation logic to reference the instance of each line object directly

15.7.3 Memory management

pp.8 LHS, Spark provides three options for storage of persistent RDDs

- in-memory storage as deserialized Java objects, which provides the fastest performance because the Java VM can access each RDD element natively
- in-memory storage as serialized data, which lets users choose a more memory-efficient representation than Java object graphs when space is limited at the cost of lower performance
- on-disk storage, which is useful for RDDs that are too large to keep in RAM but costly to recompute on each use

pp.8 RHS, to manage the limited memory available we use an LRU eviction policy at the level of RDDs, finally each instance of Spark on a cluster currently has its own separate memory space

15.7.4 Support for checkpointing

pp.8 RHS, in general checkpointing is useful for RDDs with long lineage graphs containing wide dependencies, Spark currently provides an API for checkpointing (a REPLICATE flag to persist) but leaves the decision of which data to checkpoint to the user, finally note that the read-only nature of RDDs makes them simpler to checkpoint than general shared memory, because consistency is not a concern, RDDs can be written out in the background

15.8 Evaluation

pp.9 LHS, Spark outperforms Hadoop by up to 20x, the speedup comes from avoiding I/O and deserialization costs by storing data in memory as Java objects

15.8.1 Iterative machine learning applications

pp.10 LHS, the differences between the text and binary input indicate the parsing overhead, even when reading from an in-memory file converting the pre-parsed binary data into Java objects incurs overhead

15.8.2 PageRank

pp.10 LHS, controlling the partitioning of the RDDs to make it consistent across iterations improved the speedup, the results also scaled nearly linearly to 60 nodes

15.8.3 Fault recovery

pp.10 RHS, with a checkpoint-based fault recovery mechanism recovery would likely require rerunning at least several iterations depending on the frequency of checkpoints, furthermore the system would need to replicate the application's 100GB working set across the network, in contrast the lineage graphs for the RDDs in our examples were all less than 10KB in size

15.9 Discussion

15.9.1 Expressing existing programming models

MapReduce: flatMap + groupByKey or reduceByKey if there is a combiner

Pregel: Pregel applies the same user function to all the vertices on each iteration, thus we can store the vertex states for each iteration in an RDD and perform a bulk transformation flatMap to apply this function and generate an RDD of messages, we can then join this RDD with the vertex states to perform the message exchange

pp.12 RHS, the immutability of RDDs is not an obstable because one can create multiple RDDs to represent versions of the same dataset, the reason why previous frameworks have not offered the same level of generality is that these systems explored specific problems that MapReduce and Dryad do not handle well without observing that the common cause of these problems was a lack of data sharing abstractions

15.9.2 Leveraging RDDs for debugging

pp.12 RHS, by logging the lineage of RDDs created during a job one can (1) reconstruct these RDDs later and let the user query them interactively and (2) re-run any task from the job in a single-process debugger by recomputing the RDD partitions it depends on, this approach adds virtually zero recording overhead because only the RDD lineage graph needs to be logged

15.10 References

delay scheduling: M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, *Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling*, in EuroSys '10, 2010

16 Lesson #13 Challenges and Solutions for Fast Remote Persistent Memory Access

16.1 The Publication Details

Author: Anuj Kalia, David Andersen, Michael Kaminsky

Title: Challenges and Solutions for Fast Remote Persistent Memory Access

Journal: SoCC 20: Proceedings of the 11th ACM Symposium on Cloud Computing, October 2020, pp.105—119

DOI: `https://www.pdl.cmu.edu/ftp/NVM/kalia-SoCC20.pdf`

16.2 Abstract

pp.1 LHS, compared to DRAM we find that non-volatile main memory DIMMs (NVMMs; pp.1 RHS, we use the term NVMM to refer to DIMMs with persistent media, excluding DRAM-based approaches to provide non-volatile memory such as battery-backed DRAM) have distinctive fundamental properties that pose unique challenges for networked access to NVMM both from the NIC and the CPU, we show that much of the challenges in efficient access to remote NVMM arises from the fact that CPU caches are not optimized for NVMM

16.3 Introduction

pp.1 RHS, the arrival of NVMM DIMMs with sub-microsecond latency breaks the storage latency barrier that has long limited the performance of modern datacenters, with NVMMs such as Intel's Optane DC persistent memory modules (referred to as Optane DIMMs in this work) making data durable now requires less time ($\sim$ 100ns) than a datacenter network round trip ($\approx 2\mu$s) reversing a historical trend—in the past persistent media had high latency (e.g. $\approx 10\mu$s for the fastest SSDs) and systems designed to operate at near-network latencies avoided syncing data to stable storage on the critical path of requests often sacrificing consistency guarantees
pp.2 LHS, in addition to NVMM's well-understood distinctions such as higher performance for sequential accesses than random accesses and lower performance for writes than reads, we discover new distinctions that arise specifically in the networked context, these include the interplay between PCIe or DMA accesses and the NVMM's internal block size, and performance regressions that occur in particular workloads such as networked counters and timestamps, we design methods for fast remote NVMM access that work well on current hardware, our methods address both methods of remote NVMM access—one-sided RDMA access that bypasses the remote CPU as well as RPC-based access in which the remote CPU handles all NVMM access, in addition we make the following contributions

- we show that for small persistent writes to remote NVMM, RPCs have comparable latency as one-side RDMA
- we show that disabling the data direct I/O optimization, with which NIC inject data into the CPU's L3 cache instead of DRAM/NVMM, improves bandwidth of bulk RDMA writes by avoiding random accesses to NVMM that DDIO generates
- we show how RPC-based approaches can use the CPU's DMA engines to achieve 2.3x higher bulk write bandwidth
- we present as case studies two distributed systems that demonstrate the effectiveness of our techniques, we build a state machine replication (SMR) system and a networked log server

16.4 Background

16.4.1 Non-volatile main memory

pp.2 RHS, NVMM DIMMs attach to the CPU over the memory bus and aim to provide latency and bandwidth close to DRAM, Optane DIMMs provide durability with a few hundred nanoseconds of latency in commodity servers, Optane DIMMs have higher density and lower price per gigabyte than DRAM, because NVMMs attach to the memory bus they are directly accessible from DMA-capable peripherals such as NICs
pp.3 LHS, Optane DIMM includes a controller that receives 64B commands over the DDR4 bus, and translates them into 256B reads and writes to the persistent media, the on-DIMM controller in Optane DIMMs implements write-combining, coalescing 64B DDR writes into larger media 256B writes using a write-combining buffer, we use the term "in-DIMM write amplification" to refer to the ratio of bytes written to persistent media to bytes received for writing over the DDR bus, smaller values of in-DIMM write amplification indicate more efficient use of Optane DIMMs

16.4.2 High-performance networking

pp.3 LHS, modern commodity datacenter networks support single-digit microsecond round-trip latency and up to 100Gbps of network bandwidth per server, we expect network latency to remain far above NVMM latency for the foreseeable future
pp.3 RHS, the two high-level methods of accessing remote NVMM are

one-sided RDMA: accessing NVMM directly from NICs via RDMA, which has the benefit of reducing or eliminating remote CPU use

remote procedure call: treat NVMM as conventional storage that clients access using RPCs, with RPCs the networking subsystems at the server and client are unaware of NVMM

we find that existing RPCs are much slower than RDMA NICs for bulk writes to NVMM, we improve RPC performance for this workload by using DMA engines present on the CPU die, we use eRPC for remote procedure calls in this work

16.5 Low-Latency Writes

pp.4 LHS, the key takeaway from the experiments in this section is that one-sided RDMA loses most of its latency advantage over RPCs for durable writes to remote NVMM

16.5.1 Persistent RDMA background

pp.4 LHS, with RDMA the required sequence of operations to write durably to remote NVMM is an RDMA write followed by an RDMA read with DDIO disabled (we term this combination an RDMA pwrite), the subsequent RDMA read from the client generates a DMA read at the server that flushes prior DMA writes from the server's NIC and PCIe buffers into the server CPU's memory hierarchy, with DDIO disabled the DMA writes go directly to the NVMM instead of the L3 cache

16.5.2 Durability guarantee of RDMA

pp.4 RHS, we designed a novel test to show that an RDMA write completed at the client may not be durable at the server, the test works by proving that such an RDMA write may not even be visible in the server's memory hierarchy, and is therefore outside the server's durable power-safe domain, therefore designers must include the RDMA read after the RDMA write if they require persistence

16.5.3 Measurements

pp.6 LHS, the takeaway from our measurements is that switching from volatile DRAM writes to durable NVMM writes greatly reduces the latency advantage of one-sided RDMA over RPCs, in addition to comparable latency as RDMA pwrites RPCs are simpler to use and can reduce round trips needed to complete distributed system operations, for these reasons we find that RPCs are well-suited to building low-latency distributed systems with NVMM

16.5.4 Future RDMA extensions

pp.6 LHS, there is ongoing work to specify and create a new RDMA primitive called "RDMA flush" for NVMM available in an RFC by Talpey et al. if the RDMA flush specified in this RFC becomes available one can use an RDMA write plus RDMA flush instead of an RDMA write plus RDMA read to achieve durability, in addition the RDMA flush primitive will allow keeping DDIO enabled system-wide because the RDMA flush implementation will handle the flushing of CPU caches
pp.6 RHS, replacing the flushing RDMA read with an RDMA flush will have lower latency than pwrites if NIC designers allow merging the RDMA flush request into the preceding RDMA write packet, however we expect that the latency reduction from using an RDMA flush instead of RDMA read will be small, and the resultant latency to be not much better than RPCs, this is because both approaches require similar network and PCIe operations on their critical path

16.5.5 Low-latency state machine replication

pp.7 LHS, using NVMM instead of DRAM adds only 1μs and 1.1μs to the median and 99th percentile latency respectively

16.6 High-Bandwidth Bulk Writes

pp.7 RHS, the key takeaways from the experiments in this section are

- for RDMA writes we find that disabling DDIO at the server improves bandwidth for bulk writes, since DDIO reduces bulk write bandwidth because it injects the sequential DMA writes into L3 cache which later evict into NVMM in near-random order, reducing performance
- for RPCs we find that existing RPC libraries provide low throughput for bulk writes to remote NVMM, this happens because CPU cores are much slower than RDMA or DMA engines at writing to NVMM, we show how RPC-based approaches can use an on-CPU DMA engine to offload the copying of volatile RPC buffers to NVMM buffers improving bandwidth by 2.3x

16.6.1 Discussion on disabling DDIO

pp.7 RHS, DDIO is an important optimization in volatile systems, so developers may wish to keep it in NVMM-based systems that do not require persistence, e.g. cases where NVMM serves as a large DRAM or in persistent applications that require weak consistency

16.6.2 Improving RDMA bandwidth

pp.8 LHS, our measurements using the Optane DIMMs' hardware counters show that the RDMA writes are inefficient because they cause a high in-DIMM write amplification of around 2x, this means that 2x more bytes are written to the DIMM's persistent media than are received over the network, this write amplification is problematic for two reasons—first it leaves little bandwidth for use by other applications, second because Optane DIMMs are expensive we expect some datacenter operators to use fewer DIMMs per server, which shifts the bottleneck from the InfiniBand network to the Optane DIMM, we found that the high in-DIMM write amplification happens because the CPU cache is not optimized for NVMM, the DDIO feature of the L3 cache turns that sequential RDMA accesses into near-random writes to the Optane DIMMs, disabling DDIO eliminates the access pattern randomization and write amplification
pp.8 RHS, existing proposals for cache eviction policies for NVMM operate at a cache line granularity, they do not handle the important case where the NVMM's block size is larger than the 64B cache line size, an efficient cache eviction policy for NVMM should account for block sizes larger than cache lines

16.6.3 DMA engine background

pp.9 LHS, an on-die DMA engine such as Intel's I/O acceleration technology (IOAT) DMA engine aims to accelerate memory copies in applications such as high-speed networking and storage, processors have included a DMA engine for over a decade but these accelerators have seen little use, one reason why developers have ignored DMA acceleration is that until recently their offered copy speed was fairly low, even lower than the basic memcpy speed of a single CPU core; pp.9 RHS, we are using the IOAT DMA engine to improve single-core RPC performance for bulk writes to remote NVMM

16.6.4 IOAT DMA microbenchmarks

persistence of DMA operations: by default the IOAT DMA engine writes data to CPU cache using a feature called direct cache access (DCA), DCA for DMA engines is similar to DDIO for NICs, we found that we can disable DCA and thereby force the engine's writes to go directly to the memory controller achieving durability in theory, although Intel does not yet officially support this approach for durable writes to NVMM with IOAT DMA the mechanism likely works because it is identical to RDMA writes with DDIO disabled

ordering of IOAT DMA operations: we found that keeping a window of multiple DMA operations in flight improves performance, even for moderately-sized copies of 4kB or more bytes pipelining multiple DMA operations improves performance by up to 3x (ref. pp.10 LHS), but pipelining more than eight DMAs did not further improve performance on our CPUs
NVMM applications often require ordered persistence i.e. a write should become persistent only after the previously issued write becomes persistent, for DMA operation pipelining to be usable in NVMM applications that require ordered persistence, the DMA engine should provide the guarantee: updates from an operation should not be visible in the CPU's memory subsystem before all updates from previous operations on the same channel are visible, the publicly available IOAT DMA documentation does not specify this guarantee

NVMM copy performance: we used the hardware counters on Optane DIMMs to diagnose the DMA engine's low performance, we found that the default IOAT DMA configuration results in a high in-DIMM write amplification of around 2x, the root cause is similar to DDIO's randomization effect in RDMA writes—by default the IOAT DMA engine writes data to CPU caches using DCA which trickles down to the Optane DIMMs in semi-random order, disabling DCA for the IOAT DMA engine increases peak throughput by 2x, with DCA enabled write amplification wastes almost half the bandwidth and DMA with DCA enabled does not provide durability

16.6.5 Optimizing RPCs with DMA

pp.10 RHS, we considered adding DMA support inside ePRC by using DMA to directly copy application data from the NIC's receive ring buffers to NVMM, since the DMA engine is more efficient for larger copies we let ePRC's event loop invoke an application-level request handler to copy the de-fragmented message to NVMM asynchronously using the DMA engine; pp.11 LHS, we observe that RPC + DMA with DCA disabled provides the highest performance for 16kB and larger writes

16.7 Persistent Log

pp.11 LHS, in a straightforward log design the log consists of two parts—a log buffer and an 8B counter that holds the number of bytes written to the log buffer, to append data to the log we first issue a durable write to the log buffer followed by a durable write to the counter

16.7.1 Diagnosis: cache line invalidation

pp.11 RHS, the actual reason for poor performance of repeated writes to the same cache line is that CPU caches are not optimized for NVMM, contrary to the expected behavior and common understanding clwb invalidates and flushes the target cache line, repeated invalidation of the same cache line results in poor performance

16.7.2 Rotating counter

pp.12 LHS, we designed a "rotating counter" that avoids repeated writes to the same memory location by spreading writes to multiple memory blocks, it internally uses 10 contiguous 256B blocks in NVMMs, other values of the number of contiguous blocks or the size of each block showed worse performance, after 10 updates we wrap around to the first block, during normal operation we maintain the counter's latest update in volatile memory, during failure recovery we restore the counter's state by reading from the persistent memory file and taking the maximum update value across the 10 chunks

16.7.3 Extension to rotating registers

pp.12 LHS, taking the maximum value among the 10 chunks does not work if we wish to also support decrements, we created a novel method to extend our approach by removing this limitation, our method works by using XOR instead of MAX as the recovery function

16.7.4 End-to-end performance

pp.12 RHS, we evaluate the benefit of the rotation technique described above in a networked environment, for up to 512B appends using a rotating counter improves the append rate by 80–90%

16.8 Related Work

pp.13 LHS, our work shows that we must revisit this longstanding RDMA vs. RPC debate for NVMM, our rotation technique applies to applications other than logging such as counters and timestamps

17 Lesson #13 LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation

17.1 The Publication Details

Author: Yizhou Shan, Yutong Huang, Yilun Chen, Yiying Zhang

Title: LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation

Journal: 13th USENIX Symposium on Operating Systems Design and Implementation, October 2018, pp.69

DOI: `https://www.usenix.org/system/files/osdi18-shan.pdf`

17.2 Introduction

- pp.1 RHS, multi-kernels still assume local accesses to hardware resources in a monolithic machine and their message passing is over local buses instead of a general network; pp.2 LHS, a splitkernel breaks traditional operating system functionalities into loosely-coupled *monitors* each running at and managing a hardware component, monitors in a splitkernel can be heterogeneous and can be added, removed, and restarted dynamically without affecting the rest of the system, each splitkernel monitor operates locally for its own functionality and only communicates with other monitors when there is a need to access resources there, there are only two global tasks in a splitkernel—(1) orchestrating resource allocation across components and (2) handling component failure

- pp.2 LHS, we choose not to support coherence across different components in a splitkernel because all monitors in a splitkernel communicate with each other via network messaging only, and with our targeted scale explicit message passing is much more efficient in network bandwidth consumption than the alternative of implicitly maintaining cross-component coherence

- pp.2 LHS, LegoOS is a distributed OS that apppears to applications as a set of virtual servers called *vNodes*, a vNode can run on multiple processor, memory, and storage components and one component can host resources for multiple vNodes, LegoOS cleanly separates OS functionalities into three types of monitors—(1) process monitor (2) memory monitor (3) storage monitor, LegoOS monitors share no or minimal states and use a customized RDMA-based network stack to communicate with each other

- pp.2 LHS, the biggest challenge and our focus in building LegoOS is the separation of processor and memory and their management, modern processors and OSes assume all hardware memory units including main memory, page tables, and TLB are local, LegoOS moves all memory hardware units to the disaggregated memory components and organizes all levels of processor caches as virtual caches that are accessed using virtual memory addresses, to improve performance LegoOS uses a small amount of DRAM organized as a virtual cache below current last-level cache; pp.2 RHS, LegoOS uses a novel two-level distributed virtual memory space management mechanism

17.3 Disaggregate Hardware Resource

17.3.1 Limitations of monolithic servers

inefficient resource utilization: one of the main reasons for resource underutilization in these production clusters is the constraint that CPU and memory for a job have to be allocated from the same physical machine

poor hardware elasticity: it is difficult to add, move, remove, or reconfigure hardware components after they have been installed in a monolithic server

coarse failure domain: when a hardware component within a server fails, the whole server is often unusable and applications running on it can all crash

bad support for heterogeneity: the monolithic server model tightly couples hardware devices with each other and with a motherboard, as a result making new hardware devices work with existing servers is a painful and lengthy process, moreover datacenters often need to purchase new servers to host certain hardware

17.3.2 Hardware resource disaggregation

pp.3 RHS, three hardware trends are making resource disaggregation feasible in datacenters

1. network speed has grown by more than an order of magnitude, with main memory bus facing a bandwidth wall future network bandwidth (at line rate) is even projected to exceed local DRAM bandwidth

2. network interfaces are moving closer to hardware components, as a result hardware devices will be able to access network directly without the need to attach any processors

3. hardware devices are incorporating more processing power allowing application and OS logics to be offloaded to hardware, on-device processing power will enable system software to manage disaggregated hardware components locally

17.3.3 OSes for resource disaggregation

pp.4 LHS, running OSes on processors and access memory, storage, and other hardware resources remotely misses the opportunity of exploiting device-local computation power and makes processors the single point of failure, multi-kernel solutions still run in a single server and all access some common hardware resources in the server, moreover they do not manage distributed resources or handle failures in a disaggregated cluster
pp.4 LHS, a complete disaggregation of processor, memory, and storage means that when managing one of them, there will be no local accesses to the other two

17.4 The Splitkernel OS Architecture

pp.4 RHS, the four key concepts of the splitkernel architecture are

split OS functionalities: splitkernel breaks traditional OS functionalities into monitors, each monitor manages a hardware component, virtualizes and protects its physical resources, for each monitor to operate on its own with minimal dependence on other monitors we use a stateless design by sharing no or minimal states or metadata across monitors

run monitors at hardware components: we expect each non-processor hardware component in a disaggregated cluster to have a controller that can run a monitor

message passing across non-coherent components: all communication across components in a splitkernel is through network messaging, a splitkernel still retains the coherence guarantee that hardware already provides within a component, and applications running on top of a splitkernel can use message passing to implement their desired level of coherence for their data across components

global resource management and failure handling: to minimize performance and scalability bottleneck the splitkernel only involves global resource management occasionally for coarse-grained decisions, while individual monitors make their own fine-grained decisions, the splitkernel handles component failure by adding redundancy for recovery

17.5 LegoOS Design

pp.5 LHS, LegoOS' design targets three types of hardware components—processor, memory, and storage, and we call them *pComponent*, *mComponent*, and *sComponent*

17.5.1 Abstraction and usage model

pp.5 RHS, LegoOS exposes a distributed set of virtual nodes or vNode to users, from users' point of view a vNode is like a virtual machine, LegoOS protects and isolates the resources given to each vNode from others, with splitkernel's design principle of components not being coherent LegoOS does not support writable shared memory across processors

17.5.2 Hardware architecture

pp.5 RHS, our current hardware model uses CPU in pComponent, DRAM in mComponent, and SSD or HDD in sComponent

separating process and memory functionalities: LegoOS moves all hardware memory functionalities to mComponents (e.g. page tables, TLBs) and leaves only caches at the pComponent side

processor virtual cache: after moving all memory functionalities to mComponents, pComponents will only see virtual addresses and have to use virtual memory addresses to access its caches, because of this LegoOS organizes all levels of pComponent caches as virtual caches i.e. virtually-index and virtually tagged caches
a virtual cache has two potential problems commonly known as synonyms and homonyms

synonym: happens when a physical address maps to multiple virtual addresses as a result of memory sharing across processes, and the update of one virtual cache line will not reflect to other lines that share the data, since LegoOS does not allow writable inter-process memory sharing it will not have the synonym problem

homonym: happens when two address spaces use the same virtual address for their own different data, LegoOS solves homonyms by storing an address space ID (ASID) with each cache line and differentiate a virtual address in different address spaces using ASIDs

separating memory for performance and for capacity: many modern datacenter applications exhibit strong memory access temporal locality, with good memory-access locality we propose to leave a small amount of memory at each pComponent and move most memory across the network we propose to organize pComponents' DRAM/HBM as cache rather than main memory for a clean separation of processes and memory functionalities, we place this cache under the current processor last-level cache (LLC) and call it an extended cache or *ExCache*, ExCache serves as another layer in the memory hierarchy between LLC and memory access across the network, ExCache is a virtual inclusive cache, the hit path of ExCache is handled purely by hardware—the hardware cache controller maps a virtual address to an ExCache set, fetches and compares tags in the set, and on a hit fetches the hit ExCache line, and we use LegoOS to handle the miss path of ExCache, finally we use a small amount of DRAM/HBM at pComponent for LegoOS' own kernel data usages, accessed directly with physical memory addresses and managed by LegoOS, LegoOS ensures that all its own data fits in this space to avoid going to mComponents
with our design pComponents do not need any address mappings since LegoOS accesses all pComponent-side DRAM/HBM using physical memory addresses, another benefit of not handling address mapping at pComponents and moving TLBs to mComponents is that pComponents do not need to access TLB or suffer from TLB misses

17.5.3 Process management

17.5.3.1 process management and scheduling

pp.7 LHS, after assigning a thread to a core we let it run to the end with no scheduling or kernel preemption under common scenarios, LegoOS improves the overall processor utilization in a disaggregated cluster since it can freely schedule processes on any pComponents without considering memory allocation, thus we do not push for perfect core utilization when scheduling individual threads and instead aim to minimize scheduling and context switch performance overheads, only when a pComponent has to schedule more threads than its cores will LegoOS start preempting threads on a core

17.5.3.2 ExCache management

pp.7 LHS, during the pComponent's boot time LegoOS configures the set associativity of ExCache and its cache replacement policy, while ExCache hit is handled completely in hardware LegoOS handles misses in software, LegoOS currently supports two eviction policies—FIFO and LRU

17.5.3.3 supporting Linux syscall interface

pp.7 LHS, a challenge in supporting the Linux system call interface is that many Linux syscalls are associated with states—information about different Linux subsystems that is stored with each process

and can be accessed by user programs across syscalls, with LegoOS' stateless design principle each component only stores information about its own resource and each request across components contains all the information that the destination component needs to handle the request, to solve this discrepancy between the Linux syscall interface and LegoOS' design we add a layer on top of LegoOS's core process monitor at each pComponent to store Linux states and translate these states and the Linux syscall interface to LegoOS' internal interface

17.5.4 Memory management

pp.7 RHS, we use mComponents for three types of data—anonymous memory (i.e. heaps, stacks), memory-mapped files, and storage buffer caches, LegoOS lets a process address space span multiple mComponents to achieve efficient memory space utilization and high parallelism, each application process uses one or more mComponents to host its data and a *home mComponent*—an mComponent that initially loads the process, accepts and oversees all system calls related to virtual memory space management, LegoOS uses a global memory resource manager (GMM) to assign a home mComponent to each new process at its creation time

17.5.4.1 memory space management

virtual memory space management: pp.7 RHS, we propose two-level approach to manage distributed virtual memory spaces, where the home mComponent of a process makes coarse-grained high-level virtual memory allocation decisions and other mComponents perform fine-grained virtual memory allocation, at the higher level we split each virtual memory address space into coarse-grained fix-sized virtual regions or *vRegions*, each vRegion that contains allocated virtual memory addresses (an active vRegion) is owned by an mComponent, the lower level stores user process virtual memory area (vma) information such as virtual address ranges and permissions in *vma trees* with each node in the tree being on vma, a user-perceived virtual memory range can split across multiple mComponents but only one mComponent owns a vRegion, vRegion owners perform the actual virtual memory allocation and vma tree set up; pp.8 LHS, the pComponent directly sends memory access requests to the owner of the vRegion where the memory access falls into

physical memory space management: pp.8 LHS, each mComponent manages the physical memory allocation for data that falls into the vRegion that it owns

17.5.4.2 optimization on memory accesses

pp.8 RHS, we soon found that a large performance overhead in running real applications is caused by filling empty ExCache i.e. cold misses, the basic idea of reducing the performance overhead of cold misses is simple—since the initial content of anonymous memory (non-file-backed memory) is zero, LegoOS can directly allocate a cache line with empty content in ExCache for the first access to anonymous memory instead of going to mComponent (we call such cache lines *p-local lines*), when a p-local cache line becomes dirty and needs to be flushed, the process monitor sends it to its owner mComponent which then allocates physical memory space and establishes a virtual-to-physical memory mapping, essentially LegoOS delays physical memory allocation until write time

17.5.5 Storage management

pp.8 RHS, LegoOS supports a hierarchical file interface that is backward compatible with POSIX through its vNode abstraction, to cleanly separate storage functionalities LegoOS uses a stateless storage server design where each I/O request to the storage server contains all the information needed to fulfill this request, while LegoOS supports a hierarchical file use interface, internally LegoOS storage

monitor treats (full) directory and file paths just as unique names of a file, from our observation most datacenter applications only have a few hundred files or less, thus a simple hash table for a whole vNode is sufficient to achieve good lookup performance, using a non-hierarchical file system implementation largely reduces the complexity of LegoOS' file system, making it possible for a storage monitor to fit in storage devices controllers that have limited processing power
pp.9 LHS, LegoOS places the storage buffer cache at mComponents rather than at sComponents because sComponents can only host a limited amount of internal memory, for simplicity and to avoid coherence traffic we currently place the buffer cache of one file under one mComponent, this mComponent will look up the buffer cache and returns the data to pComponent on a hit, on a miss mComponent will forward the request to the sComponent that stores the file

17.5.6 Global resource management

pp.9 LHS, LegoOS uses a two-level resource management mechanism, at the higher level LegoOS uses three global resource managers for process, memory, and storage resources, GPM, GMM, and GSM, at the lower level each monitor can employ its own policies and mechanisms to manage its local resources; pp.9 RHS, also note that LegoOS allocates hardware resources only on demand i.e. when applications actually create threads or access physical memory

17.5.7 Reliability and failure handling

pp.9 RHS, we focus on providing memory reliability by handling mComponent failure in the current version of LegoOS because

- since modern datacenter applications already provide reliability to their distributed storage data and the current version of LegoOS does not split a file across sComponent, we leave providing storage reliability to applications
- since LegoOS does not split a process across pComponents the chance of a running application process being affected by the failure of a pComponent is similar to one affected by the failure of a processor in a monolithic server, we currently do not deal with pComponent failure

pp.9 RHS, we use one primary mComponent, one secondary mComponent, and a backup file in sComponent for each vma, the primary stores all memory data and metadata, LegoOS maintains a small append-only log at the secondary mComponent and also replicates the vma tree there, when pComponent flushes a dirty ExCache line LegoOS sends the data to both primary and secondary in parallel, in the background the secondary mComponent flushes the backup log to a sComponent which writes it to an append-only file

17.6 LegoOS Implementation

17.6.1 Hardware emulation

pp.10 LHS, since there is no real resource disaggregation hardware, we emulate disaggregated hardware components using commodity servers by limiting their internal hardware usages

17.6.2 Network stack

pp.10 LHS, we implemented three network stacks in LegoOS, the first is a customized RDMA-based RPC framework we implemented based on LITE on top of the Mellanox mlx4 InfiniBand driver; pp.10 RHS, the second network stack is our own implementation of the socket interface directly on RDMA, the final stack is a traditional socket TCP/IP stack we adapted from lwip

17.6.3 Processor monitor

pp.10 RHS, we reserve a contiguous physical memory region during kernel boot time and use fixed ranges of memory in this region as ExCache, we organize ExCache into virtually indexed sets with a configurable set associacitivty, since the only chance we can trap to OS in x86 is at page fault time we use paged memory to emulate ExCache, an ExCache miss causes a page fault and traps to LegoOS, to minimize the overhead of context switches we use the application thread that faults on a ExCache miss to perform ExCache replacement
pp.11 LHS, LegoOS maintains an approximate LRU list for each ExCache set and uses a background thread to sweep all entries in ExCache and adjust LRU lists, LegoOS supports two EXCache replacement policies—FIFO and LRU

17.6.4 Memory monitor

pp.11 LHS, we use regular machines to emulate mComponents by limiting usable cores to a small number (2 to 5), we dedicate one core to busy poll network requests and the rest for performing memory functionalities, our current implementation of memory monitor is purely in software, and we use hash tables to implement the virtual-to-physical address mappings

17.6.5 Storage monitor

pp.11 LHS, we chose a simple implementation of building storage monitor on top of the Linux *vfs* layer as a loadable Linux kernel module

17.7 Evaluation

17.7.1 Micro- and macro-benchmark results

memory performance: pp.12 LHS, bypassing mComponent accesses with p-local lines significantly improves memory access performance, the difference between p-local and Linux demonstrates the overhead of trapping to LegoOS kernel and setting up ExCache

storage performance: pp.12 LHS, our result show that Linux's performance is determined by the SSD's read/write bandwidth, LegoOS' random read performance is close to Linux since network cost is relatively low compared to the SSD's random read performance

PARSEC results: pp.12 RHS, LegoOS performs worse with higher workload threads, because the single worker thread at the mComponent becomes the bottleneck to achieving higher throughput

17.7.2 Application performance

ExCache management: pp.13 RHS, surprisingly FIFO performs better than LRU in our tests even when LRU utilizes access locality pattern, we attributed LRU's worse performance to the lock contention it incurs—the kernel background thread sweeping the ExCache locks an LRU list when adjusting the position of an entry in it, while ExCache miss handler thread also needs to lock the LRU list to grab its head

number of mComponents and replication: pp.13 RHS, when there is only one mComponent flushes and misses are all between the pComponent and this mComponent, thus enabling piggyback on every flush, however when there are multiple mComponents LegoOS can only perform piggyback when flushes and misses are to the same mComponent

17.7.3 Failure analysis

pp.14 LHS, the MTTF can be calculated using the harmonic mean of the MTTF of each device

17.8 Related Work

pp.15 LHS, LegoOS is the first OS that separates memory and process management and runs virtual memory system completely at network-attached memory devices

17.9 Discussion and Conclusion

pp.15 RHS, we found that the amount of parallel threads an mComponent can use to process memory requests largely affect application throughput, thus future developers of real mComponents can consider use large amount of cheap cores or FPGA to implement memory monitors in hardware, we also found that memory allocation policies across mComponents can largely affect application performance, however since we do not support memory data migration yet the benefit of our load-balancing mechanism is small

18 Lesson #14 Large-Scale Cluster Management at Google with Borg

18.1 The Publication Details

Author: Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, John Wilkes

Title: Large-Scale Cluster Management at Google with Borg

Journal: EuroSys 15: Proceedings of the Tenth European Conference on Computer Systems, April 2015, pp.1—17

DOI: `https://dl.acm.org/doi/10.1145/2741948.2741964`

18.2 Introduction

pp.1 LHS, Borg provides three main benefits

1. hides the details of resource management and failure handling
2. operates with very high reliability and availability
3. runs workloads across tens of thousands of machines efficiently

18.3 The User Perspective

pp.1 RHS, users submit their work to Borg in the form of *jobs* each of which consists of one or more *tasks*, each job runs in one Borg *cell*

18.3.1 The workload

pp.2 LHS, we classify higher-priority Borg jobs as "production" (prod) ones and the rest as "non-production" (non-prod), most long-running server jobs are prod, most batch jobs are non-prod

18.3.2 Clusters and cells

pp.2 LHS, the machine in a cell belong to a single *cluster*, a cluster lives inside a single datacenter building, a collection of buildings makes up a *site*, our median cell size is about 10K machines after excluding test cells

18.3.3 Jobs and tasks

pp.2 LHS, a job runs in just one cell, each task maps to a set of Linux processes running in a container on a machine, the vast majority of the Borg workload does not run inside virtual machines (VMs) because we don't want to pay the cost of virtualization; pp.2 RHS, Borg programs are statically linked to reduce dependencies on their runtime environment and structured as *packages* of binaries and data files, users operate on jobs by issuing remote procedure calls (RPCs) to Borg; pp.3 LHS, tasks can ask to be notified via a Unix SIGTERM signal before they are preempted by a SIGKILL so they have time to clean up

18.3.4 Allocs

pp.3 LHS, a Borg *alloc* is a reserved set of resources on a machine in which one or more tasks can be run, the resources of an alloc are treated in a similar way to the resources of a machine, an *alloc set* is like a job, it is a group of allocs that reserve resources on multiple machines, one or more jobs can be submitted to run in it

18.3.5 Priority, quota, and admission control

pp.3 LHS, Borg defines non-overlapping *priority bands* for different uses, including (in decreasing-priority order) monitoring, production, batch, and best effort (also known as testing or free), for this paper prod jobs are the ones in the monitoring and production bands, to eliminate most of preemption cascades we disallow tasks in the production priority band to preempt one another
pp.3 RHS, quota-checking is part of admission control not scheduling—jobs with insufficient quota are immediately rejected upon submission

18.3.6 Naming and monitoring

pp.3 RHS, Borg creates a stable "Borg name service" (BNS) name for each task that includes the cell name, job name, and task number, the BNS name also forms the basis of the task's DNS name
pp.4 LHS, a service called Sigma provides a web-based user interface (UI) through which a user can examine the state of all their jobs, a particular cell, or individual jobs and tasks
pp.4 LHS, Borg records all job submissions and task events as well as detailed pre-task resource usage information in Infrastore, a scalable read-only data store with an interactive SQL-like interface via Dremel

18.4 Borg Architecture

pp.4 LHS, a Borg cell consists of a set of machines, a logically centralized controller called the Borgmaster, and an agent process called the Borglet that runs on each machine in a cell, all components of Borg are written in C++

18.4.1 Borgmaster

pp.4 LHS, each cell's Borgmaster consists of two processes—the main Borgmaster process and a separate scheduler, the Borgmaster is logically a single process but is actually replicated five times, each replica

maintains an in-memory copy of most of the state of the cell and this state is also recorded in a highly-available, distributed, Paxos-based store on the replicas' local disks, a single elected master per cell serves both as the Paxos leader and the state mutator handling all operations that change the cell's state, a master is elected using Paxos when the cell is brought up and whenever the elected master fails
pp.4 RHS, the Borgmaster's state at a point in time is called a *checkpoint* and takes the form of a periodic snapshot plus a change log kept in the Paxos store, a high-fidelity Borgmaster simulator called Fauxmaster can be used to read checkpoint files

18.4.2 Scheduling

pp.4 RHS, when a job is submitted the Borgmaster records it persistently in the Paxos store and adds the job's tasks to the pending queue, this is scanned asynchronously by the scheduler, the scheduler algorithm has two parts

feasibility checking: pp.4 RHS, finds a set of machines that meet the task's constraints and also have enough "available" resources which includes resources assigned to lower-priority tasks that can be evicted

scoring: pp.4 RHS, takes into account user-specified preferences but is mostly driven by built-in criteria such as minimizing the number and priority of preempted tasks, picking machines that already have a copy of the task's packages, spreading tasks across power and failure domains, and packing quality including putting a mix of high and low priority tasks onto a single machine to allow the high-priority ones to expand in a load spike
pp.5 LHS, Borg originally used a variant of E-PVM for scoring, in practice E-PVM ends up spreading load across all the machines leaving headroom for load spikes, but at the expense of increased fragmentation especially for large tasks that need most of the machine, we sometimes call this "worst fit", the opposite end of the spectrum is "best fit" which tries to fill machines as tightly as possible, but the tight packing penalizes any mis-estimations in resource requirements, our current scoring model is a hybrid one
pp.5 LHS, if the machine selected by the scoring phase doesn't have enough available resources to fit the new task, Borg preempts lower-priority tasks, we add the preempted tasks to the scheduler's pending queue rather than migrate or hibernate them

pp.5 LHS, to reduce task startup time the scheduler prefers to assign tasks to machines that already have the necessary packages installed

18.4.3 Borglet

pp.5 LHS, it starts and stops tasks and manages local resources by manipulating OS kernel settings; pp.5 RHS, the Borgmaster polls each Borglet every few seconds to retrieve the machine's current state and send it any outstanding requests, for performance scalability each Borgmaster replica runs a stateless *link shard* to handle the communication with some of the Borglets, for resiliency the Borglet always reports its full state, but the link shards aggregate and compress this information by reporting only differences to the state machines to reduce the update load, a Borglet continues normal operation even if it loses contact with the Borgmaster, so currently-running tasks and services stay up even if all Borgmaster replicas fail

18.4.4 Scalability

pp.6 LHS, several things make the Borg scheduler more scalable

score caching: evaluating feasibility and scoring a machine is expensive, so Borg caches the scores until the properties of the machine or task change

equivalence classes: tasks in a Borg job usually has identical requirements and constraints, Borg only does feasibility and scoring for one task per equivalence class—a group of tasks with identical requirements

relaxed randomization: it is wasteful to calculate feasibility and scores for all the machines in a large cell, so the scheduler examines machines in a random order until it has found "enough" feasible machines to score and then selects the best within that set

18.5 Availability

pp.6 LHS, Borg reduces correlated failures by spreading tasks of a job across failure domains such as machines, racks, and power domains; pp.6 RHS, a key design feature in Borg is that already-running tasks continue to run even if the Borgmaster or a task's Borglet goes down

18.6 Utilization

18.6.1 Evaluation methodology

pp.7 LHS, due to machine heterogeneity we needed a more sophisticated metric than "average utilization", we picked *cell compaction*—given a workload we found out how small a cell it could be fitted into by removing machines until the workload no longer fitted

18.6.2 Cell sharing

pp.8 LHS, prod jobs usually reserve resources to handle rare workload spikes but don't use these resources most of the time, Borg reclaims the unused resources to run much of the non-prod work, so we need fewer machines overall

18.6.3 Large cells

pp.9 LHS, Google builds large cells, both to allow large computations to be run and to decrease resource fragmentation

18.6.4 Fine-grained resources requests

pp.9 RHS, Borg users request CPU in units of milli-cores and memory and disk space in bytes (a core is a processor hyperthread), offering a set of fixed-size containers or virtual machines although common among IaaS providers would not be a good match to our needs

18.6.5 Resource reclamation

pp.10 LHS, the Borg scheduler uses limits to calculate feasibility for prod tasks, so they never rely on reclaimed resources and aren't exposed to resource oversubscription, for non-prod tasks it uses the reservations of existing tasks so the new tasks can be scheduled into reclaimed resources
pp.10 RHS, a task that exceeds its memory limit will be the first to be preempted if resources are needed regardless of its priority, so it is rare for tasks to exceed their memory limit, on the other hand CPU can readily be throttled so short-term spikes can push usage above reservation fairly harmlessly

18.7 Isolation

18.7.1 Security isolation

pp.11 LHS, we use a Linux chroot jail as the primary security isolation mechanism between multiple tasks on he same machine

18.7.2 Performance isolation

pp.11 RHS, to help with overload and overcommitment Borg tasks have an application class or *appclass*, the most important distinction is between the *latency-sensitive* (LS) appclasses and the rest which we call *batch* in this paper, a second split is between *compressible* resources (e.g. CPU cycles, disk I/O bandwidth) that are rate-based and can be reclaimed from a task by decreasing its quality of service without killing it, and *non-compressible* resources (e.g. memory, disk space) which generally cannot be reclaimed without killing the task
pp.11 RHS, we found that the standard Linux CPU scheduler (CFS) required substantial tuning to support both low latency and high utilization, to reduce scheduling delays our version of CFS uses extended per-cgroup load history, allows preemption of batch tasks by LS tasks, and reduces the scheduling quantum when multiple LS tasks are runnable on a CPU

18.8 Related Work

pp.12 LHS, Apache Mesos splits the resource management and placement functions using an offer-based mechanism, Borg mostly centralizes these functions using a request-based mechanism that scales quite well; pp.13 LHS, Borg offers a "one size fits all" RPC interface, state machine semantics, and scheduler policy
pp.13 LHS, the high-performance computing community has al ong tradition of work in this area, however the requirements of scale, workloads and fault tolerance are different from those of Google's cells, in general such systems achieve high utilization by having large backlogs of pending work

18.9 Lessons and Future Work

18.9.1 Lessons learned: the bad

jobs are restrictive as the only grouping mechanism for tasks: in Borg it is not possible to refer to arbitrary subsets of a job, to avoid such difficulties Kubernetes rejects the job notion and instead organizes its scheduling units (pod) using labels—arbitrary key/value pairs that users can attach to any object in the system, the equivalent of a Borg job can be achieved by attaching a job:jobname label to a set of pods, but any other useful grouping can be represented too

one IP address per machine complicates things: in Borg all tasks on a machine use the single IP address of their host and thus share the host's port space, in Kubernetes every pod and service gets its own IP address allowing developers to choose ports rather than requiring their software to adapt to the ones chosen by the infrastructure, and removes the infrastructure complexity of managing ports

optimizing for power users at the expense of casual ones: Borg provides a large set of features aimed at "power users" so they can fine-tune the way their programs are run, unfortunately the richness of this API makes things harder for the "casual" user, our solution has been to build automation tools and services that run on top of Borg and determine appropriate settings from experimentation

18.9.2 Lessons learned: the good

allocs are useful: the Borg alloc abstraction spawned the widely-used logsaver pattern and another popular one in which a simple data-loader task periodically updates the data used by a web server, allocs and packages allow such helper services to be developed by separate teams, the Kubernetes equivalent of an alloc is the *pod*, Kubernetes uses helper containers in the same pod instead of tasks in an alloc

cluster management is more than task management: the applications that run on Borg benefit from many other cluster services including naming and load balancing, Kubernetes supports naming and load balancing using the *service* abstraction

introspection is vital: an important design decision in Borg was to surface debugging information to all users rather than hiding it, Kubernetes has a unified mechanism that all components can use to record events that are made available to clients

the master is the kernel of a distributed system: Borgmaster was originally designed as a monolithic system, but over time it became more of a kernel sitting at the heart of an ecosystem of services that cooperate to manage user jobs, the Kubernetes architecture goes further—it has an API server at its core that is responsible only for processing requests and manipulating the underlying state objects, the culster management logic is built as small composable micro-services that are clients of this API server

18.10 References

optimistic concurrency control: M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, *Omega: flexible, scalable schedulers for large compute clusters*, in Proc. European Conf. on Computer Systems (EuroSys), Prague, Czech Republic, 2013

19 Lesson #15 Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds

19.1 The Publication Details

Author: Kevin Hsieh et al.

Title: Gaia: Geo-Distributed Machine Learning Approaching LAN Speeds

Journal: 14th USENIX Symposium on Networked Systems Design and Implementation, March 2017, pp.629

DOI: `https://www.usenix.org/system/files/conference/nsdi17/nsdi17-hsieh.pdf`

19.2 Abstract

pp.1 LHS, our goal is this work is to develop a geo-distributed ML system that (1) employs an intelligent communication mechanism over WANs to efficiently utilize the scarce WAN bandwidth while retaining the accuracy and correctness guarantees of an ML algorithm and (2) is generic and flexible enough to run a wide range of ML algorithms without requiring any changes to the algorithms, to this end we

- introduce a new general geo-distributed ML system, *Gaia*, that decouples the communication within a data center from the communication between data centers, enabling different communication and consistency models for each
- present a new ML synchronization model, *approximate synchronous parallel* (ASP) whose key idea is to dynamically eliminate insignificant communication between data centers while still guaranteeing the correctness of ML algorithms

19.3 Introduction

pp.1 RHS, existing large-scale distributed ML systems are suitable only for data residing within a single data center
pp.2 LHS, Gaia builds on the widely used parameter server architecture that provides ML worker machines with a distributed global shared memory abstraction for the ML model parameters they collectively train until convergence to fit the input data, the key idea of Gaia is to maintain an approximately correct copy of the global ML model within each data center and dynamically eliminate any unnecessary communication between data centers, Gaia enables this by decoupling the synchronization (i.e. communication/consistency) model within a data center from the synchronization model between different data centers, this differentiation allows Gaia to run a conventional synchronization model that maximizes utilization of the more-freely-available LAN bandwidth within a data center, and a new synchronization model called *approximate synchronous parallel* (ASP) across different data centers
pp.2 LHS, ASP is based on a key finding that the vast majority of updates to the global ML model parameters from each ML worker machine are insignificant, with ASP these insignificant updates to the same parameter within a data center are aggregated until the aggregated updates are significant enough, ASP ensures all significant updates are synchronized across all model copies in a timely manner, in contrast to a state-of-the-art communication-efficient synchronization model, stale synchronous parallel (SSP) which bounds how stale a parameter can be, ASP bounds how inaccurate a parameter can be in comparison to the most up-to-date value
pp.2 RHS, we build two prototype of Gaia on top of two state-of-the-art parameter server systems, one specialized for CPUs and another specialized for GPUs, we prove that ASP provides a theoretical guarantee on algorithm convergence for a widely used ML algorithm—stochastic gradient descent

19.4 Background and Motivation

19.4.1 Distributed machine learning systems

pp.3 LHS, there are many large-scale distributed ML systems such as ones using the MapReduce abstraction, ones using the graph abstraction, and ones using the parameter server abstraction, among them the parameter server architecture provides a performance advantage over other systems for many ML applications and has been widely adopted in many ML systems, in such an architecture each parameter server keeps a shard of the global model parameters as a key-value store, and each worker machine communicates with the parameter servers to read and update the corresponding parameters, the major benefit of this architecture is that it allows ML programmers to view all model parameters as a global shared memory and leave the parameter servers to handle the synchronization
pp.3 RHS, the trade-off between fresher updates and communication overhead leads to three major synchronization models

bulk synchronous parallel (BSP): synchronizes all updates after each worker goes through its shard of data, all workers need to see the most up-to-date model before proceeding to the next iteration

stale synchronous parallel (SSP): allows the fastest worker to be ahead of the slowest worker by up to a bounded number of iterations so the fast workers may proceed with a bounded stale model

total asynchronous parallel (TAP): all workers keep running based on the results of best-effort communication

both BSP and SSP guarantee algorithm convergence while there is no such guarantee for TAP, most state-of-the-art parameter servers implement both BSP and SSP

19.5 Our Approach: Gaia

19.5.1 Gaia system overview

pp.5 LHS, Gaia is built on top the popular parameter server architecture; pp.5 RHS, worker machines and parameter servers within a data center synchronize with each other using the conventional BSP or SSP models

19.5.2 Study of update significance

pp.6 LHS, not all model parameters converge to their optimal value within the same number iterations—a property called *non-uniform convergence*

19.5.3 Approximate synchronous parallel

pp.6 LHS, in this model a parameter server shares only the significant updates with other data centers and ASP ensures that these updates can be seen by all data centers in a timely fashion, ASP achieves this goal by using three techniques:

the significance filter: ASP takes two inputs from an ML programmer to determine whether or not an update is significant, they are (1) a significance function and (2) an initial significance threshold, the significance function returns the significance of each update, to ensure that the algorithm can converge to the optimal point, ASP automatically reduces the significance threshold over time (specifically if the original threshold is ν then the threshold at iteration t of the ML algorithm is $\nu/\sqrt{t}$)

ASP selective barrier: when a parameter server receives the significant updates at a rate that is higher than the WAN bandwidth can support, the parameter server first sends the indexes of these significant updates (as opposed to sending both the indexes and the update values together) via an ASP selective barrier to the other data centers, the receiver of an ASP selective barrier blocks its local worker from reading the specified parameters until it receives the significant updates from the sender of the barrier, this technique ensures that all worker machines in each data center are aware of the significant updates after a bounded network latency and they wait only for these updates, the worker machines can make progress as long as they do not depend on any of these parameters

ASP mirror clock: the ASP select barrier ensures that the latency of the significant updates is no more than the network latency, however in practice WAN bandwidth can fluctuate over time, we need a mechanism to guarantee that the worker machines are aware of the significant updates in time irrespective of the WAN bandwidth or latency, we use the mirror clock to provide this guarantee—when each parameter server receives all the updates from its local worker machines at the end of a clock e.g. an iteration, it reports its clock to the servers that are in charge of the

same parameters in the other data centers, when a server detects its clock is ahead of the slowest server that shares the same parameters by a predefined threshold DS (data center staleness), the server blocks its local worker machines from reading its parameters until the slowest mirror server catches up, this mechanism is similar to the concept of SSP but we use it only as the last resort to guarantee algorithm convergence

19.5.4 Summary of convergence proof

pp.6 RHS, the class we consider are ML algorithms expressed as convex optimization problems that are solved using distributed stochastic gradient descent, to prove algorithm convergence our goal is to show that the distributed execution of an ML algorithm results in a set of parameter values that are very close to the values that would be obtained under a serialized execution.

19.6 Implementation

19.6.1 Gaia system key components

pp.7 LHS, all of the key components are implemented in the parameter servers

local server: handles the synchronization between the worker machines in the same data center using the conventional BSP or SSP models

mirror server & mirror client: handle the synchronization with other data centers using our ASP model

each of these three components runs as an individual thread

19.6.2 System operations and communication

update from a worker machine: the local server then invokes the significant filter to determine whether or not the accumulated update of this parameter is significant, if it is the significant filter sends a MIRROR UPDATE request to the mirror client

messages from the significance filter: the significance filter sends out three types of messages

- it sends a MIRROR UPDATE request to the mirror client through the data queue
- it first sends an ASP BARRIER to the control queue before sending the MIRROR UPDATE when the significance filter detects that the arrival rate of significant updates is higher than the underlying WAN bandwidth that it monitors at every iteration
- it sends a MIRROR CLOCK request to the control queue at the end of each clock in the local server to maintain the mirror clock

operations in the mirror server: the mirror server handles the above three messages

MIRROR UPDATE: it applies the update to the corresponding parameter in the parameter store

ASP BARRIER: it sets a flag in the parameter store to block the corresponding parameter from being read until it receives the corresponding MIRROR UPDATE

MIRROR CLOCK: it updates its local mirror clock state for each parameter server in other data centers and enforces the predefined clock difference threshold DS

19.6.3 Tuning of significance thresholds

pp.8 LHS, the user of Gaia can specify two different goals for Gaia: (1) speed up algorithm convergence by fully utilizing the available WAN bandwidth and (2) minimize the communication cost on WANs, to achieve either of these goals the significance filter maintains two significance thresholds and dynamically tunes these thresholds

hard significance threshold: updates whose significance is above the hard significance threshold are guaranteed to be sent to other data centers, the purpose is to guarantee ML algorithm convergence

soft significance threshold: updates whose significance is above the soft significance threshold are sent in a best-effort manner, if the goal is to speed up algorithm convergence (the first goal) then the system lowers the soft significance threshold whenever there is underutilized WAN bandwidth; if the goal is to minimize the WAN communication costs (the second goal) then the soft significance threshold is not activated, the purpose is to use underutilized WAN bandwidth to speed up convergence

pp.8 LHS, a simple and conservative threshold (such as 1%–2%) is likely to work in most cases, this is because most ML algorithms initialize their parameters with random values and make large changes to their model parameters at early phases, thus they are more error tolerant at the beginning

19.6.4 Overlay network and hub

pp.8 LHS, this broadcast-based communication could limit the scalability of Gaia when we deploy Gaia to many data centers, to make Gaia more scalable with more data centers we use the concept of overlay networks, the WAN bandwidth between geographically-close regions is much higher than that between distant regions, in light of this Gaia supports having geographically-close data centers form a data center group, servers in a data center group send their significant updates only to the other servers in the same group, each group has hub data centers that are in charge of aggregating all the significant updates within the group and sending to the hubs of the other groups, each data center group can designate different hubs for communication with different data center groups so the system can utilize more links within a data center group

19.7 Methodology

19.7.1 Performance metrics and algorithm convergence criteria

pp.9 LHS, we use the following algorithm convergence criterion—if the value of the objective function in an algorithm changes by less than 2% over the course of 10 iterations, we declare that the algorithm has converged

19.7.2 Cost analysis

pp.10 RHS, cost is divided into three components: (1) the cost of machine time spent on computation (2) the cost of machine time spent on waiting for networks (3) the cost of data transfer across different data centers

19.8 Related Work

19.8.1 Non-uniform convergence in ML algorithms

pp.12 LHS, in many ML algorithms not all model parameters converge to their optimal value within the same number of computation iterations, several systems exploit this property to improve the algorithm

convergence speed e.g. by prioritizing the computation of important parameters, communicating the important parameters more aggressively, or sending fewer updates with user-defined filters

19.8.2 approximate queries on distributed data

pp.12 RHS, in the database community there is substantial prior work that explores the trade-off between precision and performance for queries on distributed data over WANs, the idea is to reduce communication overhead of moving up-to-date data from data sources to a query processor by allowing some user-defined precision loss for the queries

19.9 References

parameter server: A. Ahmed, M. Aly, J. Gonzalez, S. M. Narayanamurthy, and A. J. Smola, *Scalable inference in latent variable models*, in WSDM, 2012

stale synchronous parallel model: Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, More effective distributed ML via a stale synchronous parallel parameter server, in NIPS 2013

20 Lesson #16 Practical Byzantine Fault Tolerance

20.1 The Publication Details

Author: Miguel Castro and Barbara Liskov

Title: Practical Byzantine Fault Tolerance

Journal: 3rd USENIX Symposium on Operating Systems Design and Implementation, February 1999

DOI: `https://pmg.csail.mit.edu/papers/osdi99.pdf`

20.2 Summary

assumption: there are

- pp.2 LHS, we assume an asynchronous distributed system
- pp.2 LHS, we use a Byzantine failure model i.e. faulty nodes may behave arbitrarily
- pp.2 LHS, we assume independent node failures
- pp.2 LHS, we assume that the adversary cannot delay correct nodes indefinitely, we also assume that the adversary and the faulty nodes it controls are computationally bound so that with very high probability it is unable to subvert the cryptographic techniques mentioned above
- pp.3 LHS, for simplicity we assume $|\mathcal{R}| = 3f+1$ where f is the maximum number of replicas that may be faulty, although there could be more than $3f+1$ replicas the additional replicas degrade performance without providing improved resiliency
- pp.4 LHS, we assume that the client waits for one request to complete before sending the next one
- pp.4 RHS, we assume that the the strength of message digests ensures that the probability that $m \neq m'$ and $D(m) = D(m')$ is negligible

20.3 Introduction

pp.1 LHS, this paper presents a new practical algorithm for state machine replication that tolerates Byzantine faults, the algorithm offers both liveness and safety provided at most $\lfloor \frac{n-1}{3} \rfloor$ out of a total of n replicas are simultaneously faulty, the algorithm works in asynchronous systems like the internet
pp.1 LHS, most earlier work either concerns techniques designed to demonstrate the theoretical feasibility that are too inefficient to be used in practice, or assume synchrony i.e. relies on known bounds on message delays and process speeds, our algorithm uses only one message round trip to execute read-only operations and two to execute read-write operations, also it uses an efficient authentication scheme based on message authentication codes during normal operation, public-key cryptography is used only when there are faults

20.4 System Model

pp.2 LHS, we use cryptographic techniques to prevent spoofing and replays and to detect corrupted messages, our messages contain public-key signatures, message authentication codes, and message digests produced by collision-resistant hash functions, we denote a message m signed by node i as $\langle m \rangle_{\sigma_i}$ and the digest of message m by $D(m)$, we follow the common practice of signing a digest of a message and appending it to the plaintext of the message rather than signing the full message, all replicas know the others' public keys to verify signatures

20.5 Service Properties

pp.2 RHS, our algorithm can be used to implement any deterministic replicated service with a state and some operations which are not restricted to simple reads or writes of portions of the service state, the algorithm provides both safety and liveness assuming no more than $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty

safety: safety means that the replicated service satisfies linearizability, it behaves like a centralized implementation that executes operations atomically one at a time, safety requires the bound on the number of faulty replicas because a faulty replica can behave arbitrarily, safety is provided regardless of how many faulty clients are using the service—all operations performed by faulty clients are observed in a consistent way by non-faulty clients, the algorithm does not rely on synchrony to provide safety
the safety property is insufficient to guard against faulty clients, however we limit the amount of damage a faulty client can do by providing access control, the algorithm also ensures that the effects of access revocation operations are observed consistently by all clients

liveness: the algorithm relies on synchrony to provide liveness, it guarantee liveness—clients eventually receive replies to their requests—provided at most $\lfloor \frac{n-1}{3} \rfloor$ replicas are faulty and $delay(t)$ does not grow faster than t indefinitely, where $delay(t)$ is the time between the moment t when a message is sent for the first time and the moment when it is received by its destination assuming the sender keeps retransmitting the message until it is received, this is a rather weak synchrony assumption that is likely to be true in any real system provided network faults are eventually repaired

pp.3 LHS, the resiliency of our algorithm is optimal—$3f + 1$ is the minimum number of replicas that allow an asynchronous system to provide the safety and liveness properties when up to f replicas are faulty, the algorithm does not address the problem of fault-tolerant privacy—a faulty replica may leak information to an attacker, it is not feasible to offer fault-tolerant privacy in the general case

20.6 The Algorithm

pp.3 LHS, our algorithm is a form of state machine replication, the service is modeled as a state machine that is replicated across different nodes in a distributed system, each state machine replica maintains the service state and implements the service operations, the replicas move through a succession of configurations called *views*, in a view one replica is the *primary* and the others are *backups*, views are numbered consecutively, the primary of a view is replica p such that $p = v \bmod |\mathcal{R}|$ where v is the view number, view changes are carried out when it appears that the primary has failed
pp.3 RHS, like all state machine replication techniques we impose two requirements on replicas—they must be deterministic and they must start in the same state, given these two requirements the algorithm ensures the safety property by guaranteeing that all non-faulty replicas agree on a total order for the execution of requests despite failures

20.6.1 The client

pp.3 RHS, a client c requests the execution of state machine operation o by sending a $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ message to the primary, timestamp t is used to ensure exactly-once semantics for the execution of client requests, timestamps for c's requests are totally ordered, each message sent by the replicas to the client includes the current view number, a client sends a request to what it behaves is the current primary using a point-to-point message, the primary atomically multicasts the request to all the backups, a replica sends the reply to the request directly to the client, the reply has the form $\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i}$ where v is the current view number, i is the replica number, and r is the result of executing the requested operation
pp.3 RHS, the client waits for $f + 1$ replicas with valid signatures from different replicas and with the same t and r before accepting the result r since at most f replicas can be faulty, if the client does not receive relies soon enough, it broadcasts the request to all replicas, if the request has already been processed the replicas simply re-send the reply (replicas remember the last reply message they sent to each client), otherwise if the replica is not the primary it relays the request to the primary, if the primary does not multicast the request to the group, it will eventually be suspected to be faulty

20.6.2 Normal-case operation

pp.4 LHS, the state of each replica includes (1) the state of the service (2) a message log containing messages the replica has accepted (3) an integer denoting the replica's current view
pp.4 LHS, when the primary p receives a client request m it starts a three-phase protocol to atomically multicast the request to the replicas, the three phases are pre-prepare, prepare, and commit

pre-prepare: the primary assigns a sequence number n to the request, multicasts a pre-prepare message with m piggybacked to all the backups, and append the message to its log, the message has the form $\langle \langle \text{PRE-PREPARE}, v, n, d \rangle_{\sigma_p}, m \rangle$ where d is request message m's digest, requests are not included in pre-prepare messages to keep them small which allows us to use a transport optimized for small messages for protocol messages and a transport optimized for large messages for requests, a backup accepts a pre-prepare message provided that

- the signatures and the digest are correct, and the view number matches
- it has not accepted a pre-prepare message from view v and sequence number n containing a different digest
- the sequence number in the pre-prepare message is between a low water mark h and a high water mark H, which prevents a faulty primary from exhausting the space of sequence numbers by selecting a very large one

prepare: if backup i accepts the $\langle\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_p}, m\rangle$ message it enters the prepare phase by multicasting a $\langle \text{PREPARE}, v, n, d, i\rangle_{\sigma_i}$ message to all other replicas and adds both messages to its log, otherwise it does nothing, a replica including the primary accepts prepare messages and adds them to its log provided that

- the signature is correct, and the view number matches
- the sequence number is between h and H

we define the predicate $prepared(m, v, n, i)$ to be true if and only if replica i has inserted in its log, the replicas verify whether the prepares match the pre-prepare by checking that they have the same view, sequence number, and digest

commit: replica i multicasts a $\langle \text{COMMIT}, v, n, D(m), i\rangle_{\sigma_i}$ to the other replicas when $prepared(m, v, n, i)$ becomes true, this starts the commit phase, replicas accept commit messages and insert them in their log provided that

- the signature is correct, and the view number matches
- the sequence number is between h and H

we define the *committed* and *committed-local* predicates as follows

- $committed(m, v, n)$ is true if and only if $prepared(m, v, n, i)$ is true for all i in some set of $f + 1$ non-faulty replicas
- $committed\text{-}local(m, v, n, i)$ is true if and only if $prepared(m, v, n, i)$ is true, and i has accepted $2f + 1$ commits including its own from different replicas that match the pre-prepare for m i.e. having the same view, sequence number, and digest

- the pre-prepare and prepare phases guarantee that non-faulty replicas agree on a total order for the requests within a view, more precisely they ensure the following invariant—if $prepared(m, v, n, i)$ is true then $prepared(m', v, n, j)$ is false for any non-faulty replica j including $i = j$ and any m' such that $D(m') \neq D(m)$

- the commit phase ensures the following invariant—if $committed\text{-}local(m, v, n, i)$ is true for some non-faculty i then $committed(m, v, n)$ is true, this invariant and the view-change protocol described below ensure that requests that non-faulty replicas agree on a total order for the requests that commit locally across views, and that any request that commits locally at a non-faulty replica will commit at $f + 1$ or more non-faulty replicas eventually

pp.5 LHS, each replica i executes the operation requested by m after $committed\text{-}local(m, v, n, i)$ is true and i's state reflects the sequential execution of all requests with lower sequence numbers, this ensures that all non-faulty replicas execute requests in the same order as required to provide the safety property, note that we do not rely on ordered message delivery and therefore it is possible for a replica to commit requests out of order, replicas discard requests whose timestamp is lower than the timestamp in the last reply they sent to the client to guarantee exactly-once semantics

20.6.3 Garbage collection

pp.5 RHS, for the safety condition to hold messages must be kept in a replica's log until the proof that the requests have been executed by at least $f + 1$ non-faulty replicas, these proofs are generated periodically—when a request with a sequence number divisible by some constant (e.g. 100) is executed, we will refer to the sates produced by the execution of these requests as *checkpoints* and we will say

that a checkpoint with a proof is a *stable checkpoint*, a replica maintains several logical copies of the service state—the last stable checkpoint, zero or more checkpoints that are not stable, and a current state
pp.5 RHS, when a replica i produces a checkpoint it multicasts a message $\langle \text{CHECKPOINT}, n, d, i \rangle_{\sigma_i}$ to the other replicas, each replica collects checkpoint messages in its log until it has $2f + 1$ of them including its own, these $2f + 1$ messages are the proof of correctness for the checkpoint, a checkpoint with a proof becomes stable and the replica discards all pre-prepare, prepare, and commit messages with sequence number less than or equal to n from its log, it also discards all earlier checkpoints and checkpoint messages
pp.5 RHS, the checkpoint protocol is used to advance the low and high water marks, the low water mark h is equal to the sequence number of the last stable checkpoint, the high water mark $H = h + k$ where k is big enough so that replicas do not stall waiting for a checkpoint to become stable

20.6.4 View changes

pp.5 RHS, the view-change protocol provides liveness by allowing the system to make progress when the primary fails, view changes are triggered by timeouts that prevent backups from waiting indefinitely for requests to execute; pp.6 LHS, if the timer of backup i expires in view v the backup starts a view change to move the system to view $v + 1$, it stops accepting messages other than checkpoint, view-change, and new-view messages and multicasts a $\langle \text{VIEW-CHANGE}, v + 1, n, \mathcal{C}, \mathcal{P}, i \rangle_{\sigma_i}$ message to all replicas, where n is the sequence number of the last stable checkpoint known to i, $\mathcal{C}$ is a set of $2f + 1$ valid checkpoint messages proving the correctness of that checkpoint, and $\mathcal{P}$ is a set containing a set $\mathcal{P}_m$ for each request m that prepared at i with a sequence number higher than n, when the primary p of view $v + 1$ receives $2f$ valid view-change messages for view $v + 1$ from other replicas, it multicasts a $\langle \text{NEW-VIEW}, v + 1, \mathcal{V}, \mathcal{O} \rangle_{\sigma_p}$ message to all other replicas, where $\mathcal{V}$ is a set containing the valid view-change messages received by the primary plus the view-change message for $v + 1$ the primary sent, and $\mathcal{O}$ is a set of pre-prepare messages without the piggybacked request (ref. the paper for computation details), a backup accepts a new-view message for view $v+1$ if (1) it is signed properly (2) the view-change messages it contains are valid for view $v + 1$ (3) the set $\mathcal{O}$ is correct

20.6.5 Correctness

20.6.5.1 safety

pp.6 RHS, on the one hand a request m commits locally at a non-faulty replica with sequence number n in view v only if $committed(m, v, n)$ is true, this means that there is a set $\mathcal{R}_1$ containing at least $f+1$ non-faulty replicas such that $prepared(m, v, n, i)$ is true for every replica i in the set, on the other hand any correct new-view message for view $v' > v$ contains correct view-change messages from every replica i in a set $\mathcal{R}_2$ of $2f + 1$ replicas, since there are $3f + 1$ replicas $\mathcal{R}_1$ and $\mathcal{R}_2$ must intersect in at least one replica k that is not faulty, k's view-change message will ensure that the fact that m prepared in a previous view is propagated to subsequent views

20.6.5.2 liveness

pp.7 LHS, to provide liveness replicas must move to a new view if they are unable to execute a request, but it is important to maximize the period of time when at least $2f + 1$ non-faulty replicas are in the same view, we achieve these goals by three means

- to avoid starting a view change too soon, a replica that multicasts a view-change message for view $v + 1$ waits for $2f + 1$ view-change messages for $v + 1$ and then starts its timer to expire after some time T, and $2T$ for $v + 2$

- if a replica receives a set of $f+1$ valid view-change messages from other replicas for views greater than its current view, it sends a view-change message for the smallest view in the set, this prevents it from starting the next view change too late
- a faulty replica cannot cause a view change by sending a view-change message unless it is the primary, however because the primary of view v is the replica p such that $p = v \bmod |\mathcal{R}|$ the primary cannot be faulty for more than f consecutive views

20.6.6 Non-determinism

pp.7 RHS, state machine replicas must be deterministic, therefore some mechanism to ensure that all replicas select the same value is needed, since a faulty primary might send the same incorrect value to all replicas, replicas must be able to decide deterministically whether the value is correct based on the service state, this can be accomplished by adding an extra phase to the protocol—the primary obtains authenticated values proposed by the backups, concatenates $2f+1$ of them with the associated request, and starts the three-phase protocol for the concatenated message, replicas choose the value by a deterministic computation on the $2f+1$ values and their state e.g. taking the median

20.7 Optimizations

20.7.1 Reducing communication

pp.7 RHS, we use three optimizations to reduce the cost of communication

- a client request designates a replica to send the result, all other replicas send replies containing just the digest of the result
- replicas execute a request tentatively as soon as the prepared predicate holds for the request, the client waits for $2f+1$ matching tentative replies, if it receives this many, the request is guaranteed to commit eventually, this reduces the number of message delays for an operation invocation from 5 to 4
- replicas execute the request immediately in their tentative sate after checking that the request is read-only

20.7.2 Cryptography

pp.8 LHS, we actually use digital signatures only for view-change and new-view messages which are sent rarely, and authenticate all other messages using message authentication codes (MACs), however MACs have a fundamental limitation relative to digital signatures—the inability to prove that a message is authentic to a third party, we modified our algorithm to circumvent the problem by taking advantage of specific invariants, e.g. the invariant that no two different requests prepare with the same view and sequence number at two non-faulty replicas, MACs can be computed three orders of magnitude faster than digital signatures
pp.8 RHS, each node shares a 16-byte secret session key with each replica, rather than using the 16 bytes of the final MD5 digest we use only the 10 least significant bytes, this is a variant of the secret suffix method which is secure as long as MD5 is collision resistant
pp.8 RHS, the digital signature in a reply message is replaced by a single MAC which is sufficient because these messages have a single intended recipient, the signatures in all other messages are replaced by vectors of MACs that we call *authenticators*, an authenticator has an entry for every replica other than the sender, the size of authenticators grows linearly with the number of replicas but it grows slowly—it is equal to $30 \times \lfloor \frac{n-1}{3} \rfloor$ bytes

20.8 Implementation

20.8.1 The replication library

pp.9 LHS, the client interface to the replication library consists of a single procedure *invoke* with one argument—an input buffer containing a request to invoke a state machine operation, it returns a pointer to a buffer containing the operation result, on the server side the replication code makes a number of upcalls to server procedures, point-to-point communication between nodes is implemented using UDP, and multicast to the group of replicas is implemented using UDP over IP multicast, there is a single IP multicast group for each service
pp.9 LHS, view changes can be used to recover from lost messages, but this is expensive and therefore it is important to perform retransmissions, during view changes replicas retransmit view-change messages until they receive a matching new-view message or they move on to a later view, the replication library does not implement view changes or retransmissions at present

20.8.2 BFS: a Byzantine-fault-tolerant file system

pp.9 RHS, we rely on user level *relay* processes to mediate communication between the standard NFS client and the replicas, a relay receives NFS protocol requests, calls the *invoke* procedure of our replication library, and sends the result back to the NFS client, each replica runs a user-level process with the replication library and our NFS V2 daemon which we will refer to as *snfsd*, we implemented *snfsd* using a fixed-size memory-mapped file, all the file system data structures e.g. inodes, blocks, and their free lists are in the mapped file, we rely on the operating system to manage the cache of memory-mapped file pages and to write modified pages to disk asynchronously; pp.10 LHS, we do not require synchronous writes to implement NFS V2 protocol semantics because BFS achieves stability of modified data and meta-data through replication

20.8.3 Maintaining checkpoints

pp.10 LHS, *snfsd* executes file system operations directly in the memory mapped file to preserve locality, and it uses copy-on-write to reduce the space and time overhead associated with maintaining checkpoints, *snfsd* maintains a copy-on-write bit for every 512-byte block in the memory mapped file, while executing a client request *snfsd* checks the copy-on-write bit for the block, and if it is set, stores the block's current contents and its identifier in the checkpoint record for the last checkpoint, then it overwrites the block with its new value and resets its copy-on-write bit

20.8.4 Computing checkpoint digests

pp.10 RHS, *snfsd* uses an incremental collision-resistant one-way hash function called AdHash, this function divides the state into fixed-size blocks and uses some other hash function (e.g. MD5) to compute the digest of the string obtained by concatenating the block index with the block value for each block, the digest of the state is the sum of the digests of the blocks modulo some large integer, to compute the digest for the state incrementally *snfsd* maintains a table with a hash value for each 512-byte block, this hash value is obtained by applying MD5 to the block index concatenated with the block value at the time of the last checkpoint, then it adds the new hash value to d, subtracts the old hash value from d, and updates the table

20.9 Performance evaluation

pp.10 RHS, this section evaluates the performance of our system using two benchmarks

micro-benchmark: provides a service-independent evaluation of the performance of the replication library, it measures the latency to invoke a null operation

Andrew benchmark: compare BFS with two other file systems—the NFS V2 implementation in Digital Unix (NFS-std), and BFS without replication (BFS-nr)

20.9.1 Micro-benchmark

pp.11 RHS, this micro-benchmark represents the worst case overhead for our algorithm because the operations perform no work and the unreplicated server provides very weak guarantees, most services will require stronger guarantees e.g. authenticated connections, and the overhead introduced by our algorithm relative to a server that implements these guarantees will be lower

20.9.2 Andrew benchmark

pp.12 LHS, the Andrew benchmark emulates a software development workload, it has five phases: (1) creates subdirectories recursively (2) copies a source tree (3) examines the status of all the files in the tree without examining their data (4) examines every byte of data in all the files (5) compiles and links the files
pp.12 LHS, the comparison between BFS-strict and BFS-nr shows that the overhead of Byzantine fault tolerance for this service is low, the overhead is lower than what was observed for the micro-benchmarks because the client spends a significant fraction of the elapsed time computing between operations i.e. between receiving the reply to an operation and issuing the next request, but the overhead is not uniform across the benchmark phases, the main reason for this is a variation in the amount of time the client spends computing between operations
pp.13 LHS, both versions of BFS are faster than NFS-std for operations that require the NFS implementation to ensure stability of modified file system state before replying to the client, because NFS-std achieves stability by writing modified state to disk whereas BFS achieves stability with lower latency using replication

20.10 Related Work

pp.13 LHS, most previous work on replication techniques ignored Byzantine faults or assumed a synchronous system model, furthermore our system uses view changes only to select a new primary but never to select a different set of replicas to form the new view
pp.13 LHS, both Rampart and SecureRing must exclude faulty replicas from the group to make progress, they rely on failure detectors to determine which replicas are faulty, however failure detectors cannot be accurate in an asynchronous system i.e. they may misclassify a replica as faulty, our algorithm is not vulnerable to this problem because it never needs to exclude replicas from the group

20.11 References

public-key signature: R. Rivest, A. Shamir, and L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, 21(2), 1978

message authentication code: G. Tsudik, *Message Authentication with One-Way Hash Functions*, ACMComputer Communications Review, 22(5), 1992

collision-resistant hash function: R. Rivest, *The MD5 Message-Digest Algorithm*, Internet RFC-1321, 1992

21 Lesson #17 The Computing Landscape of the 21st Century

21.1 The Publication Details

Author: Mahadev Satyanarayanan, Wei Gao, Brandon Lucia

Title: The Computing Landscape of the 21st Century

Journal: HotMobile 19: Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications, February 2019, pp.45-–50

DOI: `https://elijah.cs.cmu.edu/DOCS/satya-hotmobile2019.pdf`

21.2 A Tiered Model of Computing

pp.1 RHS, today's computing landscape is best understood by the tiered model, each tier represents a distinct and stable set of design constraints that dominate attention at that tier, there are typically many alternative implementations of hardware and software at each tier, but all of them are subject to the same set of design constraints, there is no exception of full interoperability across tiers—randomly choosing one comonent from each tier is unlikely to result in a functional system, rather there are many sets of compatible choices across tiers

21.2.1 Tier-1: elasticity, permanence and consolidation

pp.1 RHS, tier-1 represents "the cloud" in today's parlance, two dominant themes of tier-1 are *compute elasticity* and *storage permanence*, cloud computing has almost unlimited elasticity as a tier-1 data center can easily spin up servers to rapidly meet peak demand, relative to tier-1 all other tiers have very limited elasticity, in terms of archival preservation the cloud is the safest place to store data with confidence that it can be retrieved far into the future
pp.1 RHS, relative to the data permanence of tier-1 all other tiers offer more tenuous safety, getting important data captured at those tiers to the cloud is often an imperative, tier-1 exploits economies of scale to offer very low total costs of computing, as hardware costs shrink relative to personnel costs it becomes valuable to amortize IT personnel costs over many machines in a large data center, *consolidation* is thus a third dominant them of tier-1, for large tasks without strict timing, data ingress volume, or data privacy requirements, tier-1 is typically the optimal place to perform the task

21.2.2 Tier-3: mobility and sensing

pp.2 LHS, *mobility* is a defining attribute of tier-3 because it places stringent constraints on weight, size, and heat dissipation of devices that a user carries or wears, battery life is another crucial design constraint
pp.2 LHS, *sensing* is another defining attribute of tier-3, while mobile hardware continues to improve, there is always a large gap between what is feasible on a mobile device and what is feasible on a server of the same technological era, one can view this stubborn gap as a "mobility penalty"—the price one pays in performance foregone due to mobility constraints, to overcome this penalty a mobile device can offload computation over a wireless network to tier-1, IoT devices can be viewed as tier-3 devices because there is a strong incentive for them to be inexpensive, which typically implies meager processing capability and makes offloading computation to tier-1 attractive

21.2.3 Tier-2: network proximity

pp.2 LHS, extreme consolidation has two negative consequences: (1) it tends to lengthen network round-trip times (RTT) to tier-1 from tier-3 (2) the high fan-in from tier-3 devices implies high cumulative ingress bandwidth demand into tier-1 data centers, tier-2 addresses these negative consequences by creating the illusion of bringing tier-1 "closer", this achieves two things—it enables tier-3 devices to offload compute-intensive operations at very low latency, proximity also results in a much smaller fan-in between tier-3 and tier-2 than is the case when tier-3 devices connect directly to tier-1, server hardware at tier-2 is essentially the same as at tier-1 but engineered differently, instead of extreme consolidation servers in tier-2 are organized into small dispersed data centers called cloudlets, which can be viewed as "a data center in a box", the introduction of tier-2 is the essence of *edge computing*
pp.2 RHS, note that proximity here refers to network proximity rather than physical proximity, it is crucial that RTT be low and end-to-end bandwidth be high, this is achievable by using a fiber link between a wireless access point and a cloudlet, conversely physical proximity does not guarantee network proximity

21.2.4 Tier-4: longevity and opportunism

pp.3 LHS, a key driver of tier-3 is the vision of *embedded sensing* in which tiny sensing-computing-communication platforms continuously report on their environment, the challenge of cheaply maintaining tier-3 devices in the field has proved elusive because replacing their batteries or charging them is time-consuming and/or difficult, this has led to the emergence of devices that contain no chemical energy source (battery), instead they harvest incident EM energy (e.g. light or RF) to charge a capacitor which then powers a brief episode of sensing, computation and wireless transmission, the device then remains passive until the next occasion when sufficient energy can be harvested to power another episode, this modality of operation, referred to as *intermittent computing*, eliminates the need for energy-related maintenance of devices in the field, this class of devices constitutes tier-4, *longevity* of deployment combined with *opportunism* in energy harvesting are the distinctive attributes of this tier
pp.3 LHS, the most successful tier-4 devices today are RFID tags, a tier-3 device (e.g. RFID reader) provides the energy that is harvested by a tier-4 device, *immersive proximity* is thus the defining relationship between tier-4 and tier-3 devices—they have to be physically close enough for the tier-4 device to harvest sufficient energy for an episode of intermittent computation, network proximity alone is not sufficient

21.3 Using the Model

pp.3 LHS, not every distributed system will have all four tiers, each tier embodies a small set of salient properties that define the reason for the existence of that tier

tier 1: elasticity, permanence and consolidation

tier 2: network proximity to tier-3

tier 3: mobility and sensing

tier 4: longevity and opportunism

the salient attributes of a tier severely constrain its range of acceptable designs, the same reasoning also applies to software at each tier, e.g.

- tier-3 to tier-1 communication is over WAN and may involve a wireless first hop that is unreliable and/or congested, successful tier-3 software design for this context has to embody support for disconnected and weakly-connected operation

- tier-3 to tier-2 communication is expected to be LAN or WLAN quality at all times, a system composed of just those tiers can afford to ignore support for network failure

21.4 The Central Role of Energy

tier-1: (data centers) power used can add up to as much as 30MW at peak hours, current power saving techniques focus on load balancing and dynamically eliminating power peaks, power over-subscription enables more servers to be hosted than theoretically possible, leveraging the fact that their peak demands rarely occur simultaneously

tier-2: (cloudlets) cloudlets can span a wide range of form factors, power consumption can therefore vary from < 100W to several kilowatts, at this tier well-known power saving techniques such as CPU frequency scaling are applicable, energy constraints are relatively easy to meet at this tier

tier-3: (smartphones) smartphones are the dominant type of computing device at tier-3, their power consumption is below 1000mW when idle but can peak at 3500-4000mW, techniques such as frequency scaling, display optimization, and application partitioning are used to reduce power consumption
(wearables) the energy consumption of smartswitches can be usually controlled to below 100mW in stand-by mode with screen off, when the screen is on or the device is wirelessly transmitting data the energy consumption could surge to 150–200mW, various techniques have been proposed to further reduce smartwatch power consumption to < 100mW in active modes via energy-efficient storage or display management

tier 4: energy harvesting presents unique challenges—sporadic power is limited to 10^{-7} to 10^{-8} watts using e.g. RF or biological sources, a passive RFID tag consumes hundreds of nA at 1.5V, emerging wireless backscatter networking enables communication at extremely low power

21.5 A Tiered View of the Past

- pp.4 LHS, in the beginning there was only tier-1, the other tiers could not emerge until the hardware cost of computing and its physical size had dropped by many orders of magnitude, the emergence of timesharing by the late 1960s introduced elasticity to tier-1; pp.4 RHS, timesharing multiplexed the mainframe at fine granularity rather than serially reusing it, which made tier-1 appear elastic to varying numbers of users, this encapsulating ability led to the resurgence of VMs in cloud computing

- pp.4 RHS, frustration with the queueing delays of timesharing led to the emergence of personal computing, in this major shift tier-1 was completely replaced by the brand-new tier-2, an unintended consequence of the disaggregation of tier-1 into dispersed tier-2 elements was its negative impact on shared data, the disaggregation of tier-1 into dispersed tier-2 devices destroyed the mechanisms for data sharing across users, it was at this juncture that the third important attribute of tier-1, namely storage permanence, came to be recognized as crucial; pp.5 LHS, a distributed file system created the illusion that all of tier-1 storage was accessible via on-demand caching at tier-2 devices, today systems such as Dropbox and Box are modern realizations of this concept

- pp.5 LHS, the emergence of tier-3 coincided with the release of the earliest computers that were small enough to be considered portable devices, a key distinction between tier-2 and tier-3 was the stability and quality of internet connectivity, in contrast to tier-2 devices tier-3 devices typically had wireless connectivity with periods of disconnection and poor connectivity

21.6 Future Evolution

biological computer systems: a key advantage of engineered biological computing systems is their extremely high degree of parallelism, a key challenge is the lack of reliability of individual components, this can be mitigated by using the extreme parallelism for redundancy

blurring boundaries between tiers: tier boundaries are likely to blur leading to a continuum of devices with different power budgets, computing workloads and manufacturing costs, the major drivers of such blurring are advances in the manufacturing technologies, a consequence of such blurring boundaries is that the gap in capabilities between cloudlets at tier 2 and battery-powered mobile devices at tier 3 will diminish, energy harvesting will also be able to provide a much higher power budget which then allows a richer set of computing tasks being executed at tier-4 devices, consequently the transition between tier-3 and tier-4 will be much smoother

21.7 References

edge computing: Mahadev Satyanarayanan, *The Emergence of Edge Computing*, IEEE Computer 50, 1 (2017)

intermittent computing: Brandon Lucia, Vignesh Balaji, Alexei Colin, Kiwan Maeng, and Emily Ruppel, *Intermittent Computing: Challenges and Opportunities*, In Proceedings of the 2nd Summit on Advances in Programming Languages