2017-2 Distributed Systems Part2

45748 characters in 7469 words on 1242 lines

Florian Moser

February 20, 2018

1 Introduction

reasons for distributed systems
geography (large firms are active worldwide)
parallelism (multicore to speed up computation)
reliability (replication to prevent data loss)
availability (replication for fast access)

problems of distribution
coordination (consistency, agreement, consensus)
probability high that some machine in cluster is down

2 Fault-Tolerance & Paxos

2.1 definitions

node
single actor in system
total amount of nodes denoted by n

client / server
client node wants to manipulate data on server node

sending one at a time
only ever sends next message if previous message ACKed

sequence number
unique number attached to every message
allows to discover duplicates

state replication
archived by set of nodes which execute commands in same order
fundamental property

ticket
weaker form of lock, reissuable, expires if new one arrived
reissuable to deal with crashes, implement with counter

2.2 models

message passing model MPM
set of nodes, each performs computations, sends messages to all others

MPM with message loss MPMML
any message may be lost on the way to receiver
message corruption better than total loss (can use checksums)

variable message delay VMD
different transaction times for each message possible

2.3 algorithms

2.3.1 naive client-server

client sends commands to server

remarks
not robust against message loss

2.3.2 client-server with ACK

client sends commands one at a time to server
servers responds with ACK
client resends if no ACK received

remarks
include sequence numbers to avoid double execution
basis of TCP and other reliable protocols

inconsistent state possible with multiple servers
due to reordering because of VMD
proof with x+1, x*2 received at different orders

2.3.3 state replication with serializer

client sends commands one at a time to serializer
serializer forwards one at a time to servers
serializer notifies client of success

remarks
also called master-slave
serializer is single point of failure

2.3.4 two-phase protocol

(phase 1)
client asks servers for lock
(phase 2)
if (client has locks from all) then send command, return locks
else give locks back, wait and restart phase 1

remarks
applications include 2PC, 2PL, 3PL
needs to have all servers available to function

2.3.5 naive ticket protocol

(phase 1)
client asks servers for ticket
(phase 2)
if (majority replied) then
client sends command with ticket number
servers respond if ticket number is max
(phase 3) continue if majority replied
if (majority replied) then
client tells servers to execute
else
client waits and restarts phase 1

remarks
if client slow with execute, other may slip in own command

2.3.6 paxos

(init)
client has command c, ticket number t=0
server has T_max = 0, T_c = 0, C = null
(phase 1)
client increases t, sends (t,c) to server
if (t > T_max) then server updates T_max, responds with (T_store, C)
(phase 2) continue if majority replied
set c = max TStore C?? c, send propose(t, c) to same
if (t == T_max) then server sets C, T_store, answers success
(phase 3) continue if majority replied
sends execute(c) to all

remarks
clients can abort and restart at any point
randomize waiting times and send NACK for better performance
needs majority of servers up, needs trusted clients
instance of paxos decides on single command
run paxos in parallel to decide on multiple commands
may not terminate

worst-case
all clients start at same time, same timeout, same initial ticket
then they keep invalidating each other at phase 2, can't enter phase 3

improve
do not use different initial ticket numbers, leads to starvation
use exponential backoff for timeouts in phase 2, phase 3

proposal chosen is kept
chose if majority of servers accept
only can happen if majority replied in (phase 1)
therefore for all t only single c can be chosen

3 Consensus

3.1 definitions

consensus
correct nodes must decide on single value from the proposed values
agreement (1) & termination (2) & validity (3) (any-input)
nodes have no shared memory, focus on algorithms with progress

asynchronous runtime
time units (delay of single message) from start to end at worst case
cannot make assumptions about maximum delay in algorithm

configuration
fully defined system at specific point in time
includes all messages in transit & state of all nodes
initial configuration is C_0, all nodes have sent first message

univalent configuration
if decision value fixed no matter of what happens afterwards

v-valent configuration
univalent configuration for value v

bivalent
if decision value not fixed (yet)

transition T
from C_1 to C_2, is event (u,m) with node u receiving message m
in C_2, m arrived, u changed state, new messages from u in transit

configuration tree
directed tree of Cs, top is C_0, edges are all possible Ts
every algorithm has one tree per selection of input values
leaves are univalent end states
every possible path to every leave is one possible execution
if node u crashes in C remove all (u, $*$) transitions

critical configuration C
if C is bivalent but all C's below are univalent

3.2 models

asynchronous model
algorithms are event based (on receive, do ...)
no synchronized time
messages are received in finite but unbound time

3.3 proofs

there are bivalent C_0
build a = array of initial values, index of a determines number of 0
a_0 is 0-valent, a_n is 1-valent, there must be turning point a_i
node i crashes, the other nodes must terminate & decide on value

(u1, m1) (u2,m2) end in same C
as consume/produce same messages, states, they end in same C

system must reach critical configuration to terminate
assume bivalent start, always progress in bivalent configuration

no termination if crash at critical configuration
define C with t0 0-valent and t1 1-valent
must happen on same node; crash that node and remove transitions

if f $>$ 0 & deterministic algorithm then no termination
crash node with critical configuration

f $\geq$ n/2 can't archive consensus
define S1 all nodes are 1, S2 half 0/half 1, S3 all are 0
build set N and N' with each n/2 nodes, crash N'
if N is all 1, can't know if in S1 or S2
this proof sketch is useful for similar problems

consensus with f edge failures
assuming fully connected network, n$*$(n-1)/2 edges
largest f is path between remaining nodes
smallest f partitions, n-1

3.4 algorithms

3.4.1 sending ACK after ACK

receiver sends ACK after receiving message
sender confirms ACK with ACK

remarks
can never terminate, need to continue indefinitely
proof by assuming an algorithm exists, and then losing its last ACK

3.4.2 naive consensus

node broadcasts its value, waits for all others
chooses minima

remarks
does not tolerate crashes

3.4.3 randomized consensus (async)

(init)
client has v = {0,1}, round = 1, decided = false
broadcast myValue(v, round)
(propose) when majority of myValue messages received
if (all same value) then propose(v, round)
else propose(null, round)
if (decided) myValue(v, round+1) and terminate
(adapt) when majority of propose messages received
if (all same proposed) v=proposed and decided=true
elseif (at least one proposed) v=proposed
else choose v randomly
round += 1, myValue(v, round)
(restart round at propose)

remarks
progress ensured at (propose) and (adapt) for f $<$ n/2
cannot set v deterministic, proof using a partition

validity fulfilled
trivial for univalent input, else result does not matter

agreement fulfilled
terminates if decided is true, is true if all propose same value
send of propose possible if majority same myValue v received
it follows all have received at least one propose with myValue v
nodes which can't decide adapt own value and decide next round

termination fulfilled
all pick same value with probability 2ˆ-n
constant messages per round, therefore O(2ˆn)

3.4.4 shared coin (async)

broadcast coin c = 0 with 1/n, else c = 1
wait for n-f coins
put all coins in set and broadcast
wait for n-f sets
if (0-coin in any sets) decide 0 else decide 1

remarks
at most f nodes crash therefore the two waits are OK
all coins are seen by all correct nodes

3.4.5 exercise algorithms

bandwidth limitation (sync)
assume nodes with ID, no crashes, single message per round
determine leader by id = 1, leader sends value to id = 2
generalize, only log(n) rounds necessary

consensus in a grid
each node broadcast own value, resends all received values
waits till no new information received
l+1 rounds for l = w+h of grid
need only single byzantine node to deceiver corner node

4 Byzantine Agreement

4.1 definitions

byzantine
node with arbitrary behaviour, also includes malicious

byzantine agreement
finding consensus with byzantine nodes in network
agreement (1), termination (2), any-input validity (3)

f-resilient
system still operative with f byzantine nodes

any-input validity ("normal")
decision value is input of any node

correct-input validity
decision value is input of a correct node

all-same validity
if all correct nodes propose same value, this will be decided

median validity
decision value is value close to the median of correct nodes

synchronous runtime
number of rounds from start to end of worst case

bounds
upper bound if problem can be solved in that time
lower bound if problem can't be solved in less time
tight bound if lower=upper bound

4.2 models

synchronous model
operation in rounds, each round messages are sent and received

4.3 proofs

all-same validity & byzantine agreement needs n > 3
for n=3 can't differentiate friends from fiend

byzantine agreement needs f < n/3
combine nodes in 3 supernodes, then use proof for n > 3

need at least f+1 rounds to decide for minima
f times contact single neighbour which crashes afterwards
general proof also possible

paxos fails
assume two majorities which overlap in one server
this server can decide for different values in the two majorities

4.4 algorithms

4.4.1 byzantine agreement (for f=1)

node u has value x
(round 1)
send tuple (u,x) to all others, receive tuples from all others
store received tuples in set S
(round 2)
send set S to all others, receive sets U from all others
choose smallest value from tuples contained > 1 in (U + S)

remarks
in round 2 only received tuples are resend, but not own
all correct nodes will have same U
archive all-same validity with multiple occurrence of min-value

4.4.2 king algorithm (for f < n/3)

node has value x
for phase=1 till f+1
(round 1)
value(x)
(round 2)
if (#value(v) > n-f) then propose(v)
if (#propose(z) > f) then x=z
(round 3)
predefined king of phase broadcasts his value(w)
if (#propose(x) < n-f) then x=w
endfor

remarks
first adaptation needed for king
second adaptation for backups
after a correct node had been king, no changes any more
correct nodes propose only one value
at least one phase has a correct king
additionally all-same validity

4.4.3 asynchronous byzantine agreement (f < n/10)

node has value x, r = 1, decided = false
propose(x, r)
(loop) wait for n-f propose message of current round
if (#propose y > n/2 + 3f + 1) then propose(y, r+1), terminate
if (#propose y > n/2 + f + 1) then x = y
else choose x randomly
r+=1, propose (x, r)
restart (loop)

remarks
x=y only happens for single y at correct nodes

all-same validity
when than n-2f propose from correct nodes received
then n-2f > (n/2 + 3f + 1)

agreement
when (n/2 + 3f + 1) was fulfilled, then -2f worst case
but will still adopt y, and then terminate next round

termination
with probability 1/2ˆ(n-f)+1

5 Authenticated Agreement

5.1 definitions

signature
nodes can sign a message to verify origin

5.2 models

system model SM
n = 3f + 1, unbound number of clients which send requests
messages are asynchronous, have variable delay, can get lost

5.3 proofs

quorum intersection 2f + 1 in SM
intersection of two 2f+1 sets has at least one correct node

5.4 algorithms

5.4.1 byzantine agreement with authentication (sync)

primary has input, all nodes can sign messages with signature
(primary p)
if (input) send value(1)_p
decide input and terminate
(secondary u)
for i=0 till f
if (#received messages > i and value(1)_p contained) then
broadcast all received messages + value(1)_u, terminate

remarks
solves for any number of failures, signatures help detect byzantine

byzantine primary
to avoid a byzantine primary to dictate run the algorithm in parallel
need 2f+1 nodes (f < n/2) as primary, choose where #result > f+1

more than 0-1
primary always broadcast
secondary checks that only single value received from primary

5.4.2 practical byzantine fault tolerance (async)

view
integer which determines configuration state of protocol

primary, backups
for view v, node u = v mod n is the primary, others are backups

accepted messages
authenticated (signed), protocol-correct, same view messages

faulty timer
started while waiting for response from primary
trigger view change if timeout occurs

prepared certificate PC
2f+1 prepare messages (can include own)

new-view certificate NVC
2f+1 view_change messages (can include own)

remarks
signatures verify sender
in each view, there is always a primary, the others are backups
if primary is byzantine, backups can initiate a view change
correct primary chooses dense sequence numbers sn
if one node executed (request, sn) eventually all other will too
nodes collect 2f+1 confirmation messages before executing (r, sn)
if client receives f+1 replies it can assume execution
if prepare certificate obtained, no others exists

request accept
(phase 1, primary p, view v, sequence number s)
accepts request from client
send pre-prepare(v, s, r, p)_p
(phase 1, backup b)

accepts request from client and relays to primary

remarks request accept
client sends request to all servers
primary sends pre-prepare else byzantine could force view_changes
(by sending distinct pre-prepare to all nodes, faulty timers expire)

prepare & pre-prepare
(phase 2, backup b)
accept pre-prepare(v, s, r, p)_p
verify p is primary of v, verify first pre-prepare for (v,s)
send prepare(v, s, r, b)_b
(phase 3, node i)
wait for 2f prepare matching (v,s,r) (PC)
send commit(v,s,i)
wait for 2f+1 commit matching (v,s)
execute r after all lower r's have been executed
send reply to client

view change initialize
(after faulty timer at backup b has expired)
stops accepting for v
send view_change(v+1, P_i, i) with P_i are PC's already established

view change protocol
(new primary p of view v+1)
wait for 2f+1 view_change messages, put in V
O is set of all pre-prepare with PC from V, adapted for v+1
O has pre-prepare(v+1, s, null, p)_p for all s with missing PC
send new_view(v+1, V, O, p)_p to all nodes
start processing with s_max + 1 from O
(backup b of view v+1)
accept new-view(v+1, V, O, p)_p, stops accepting for v
verify p is primary, verify O correctly constructed from V
if (verify OK) then start at (phase 2) else trigger view change v+2

remarks
unique sequence numbers even across views, so each (v,r) is unique
after view_change message, faulty timer if new primary is byzantine

6 Quorum Systems

6.1 definitions

quorum
quorum is a subset of all nodes, such that any two overlap
minimal if its the smallest subset possible

access strategy
probability for each node that it is accessed

work of quorum
number of nodes in one quorum

work of access strategy
expected number of nodes accessed

work of quorum system W(Q)
work of best access strategy
sumof (p_quorum * quorum_size) for all quorums

load of node
probability that it is accessed
sumof (p_quorum) for all quorums node is part of

load of access strategy
maximum load of any node from the system

load of quorum system L(Q)
load of best access strategy

work vs load
work is what has to be done, load is how well its distributed
work is usally a big number (3), load is small (1/3)

uniform access strategy
work is amount of nodes per quorum
load is work(Q)/#quorums

failure probability F_p
p a quorum systems fails assuming nodes fail with fixed p

asymptotic failure probability afp
p when n → inf

f-resilience R(Q)
if f nodes can fail but quorum still possible

f-disseminating
assumes self-verifying messages
(1) intersection of two quorum systems contains f+1 nodes
(2) for f byzantine nodes there is quorum system without one
cannot double-spend information (1), cannot simply crash (2)
need more than 3f nodes
L ≥ sqrt(f+1 / n) because f+1 element accessed for quorum

f-masking
extension of f-disseminating because can falsify information
(1) intersection of two quorum systems contains 2f+1 nodes
(2) for f byzantine nodes there is quorum system without one
cannot falsify information (1), cannot simply crash (2)
need more than 4f nodes
L ≥ sqrt(2f+1 / n) because 2f+1 elements accessed for quorum

f-opague quorum system
ensures each quorum accesses more up-to date nodes than others
n > 5f, L(S) ≥ 0.5

s-uniform
if every quorum has exactly s elements

balanced access strategy
if load constant on all nodes of quorum system

6.2 proofs

L ≥ sqrt(1/n)
need to access a node in all minimal quorums

6.3 access strategies

6.3.1 primary copy

single node locked

remarks
W = 1, L = 1, R = 0, afp = 1 - p
good choice if failure probability is over 1/2

6.3.2 majority system

more than half of the nodes locked

remarks
W > n/2, L > 1/2, R < n/2, afp = 0

6.3.3 basic grid quorum system

assume sqrt(n) element_of natural_numbers
quorum i consists of row & column i

remarks
W = 2sqrt(n) - 1, L = 1/sqrt(n), R = sqrt(n)-1, afp = 1
each quorum has two intersections
totally order nodes

sequential locking strategy
lock nodes ordered by identifiers
release all locks if some already locked

concurrent locking strategy
priority to quorum which already has highest node locked

better systems
for example T form, size can be reduced till sqrt(n)

6.3.4 B-grid quorum system

n = d*h*r where d=#columns, h=#band, r=#rows in band
quorum has column in each band (h*r)
additionally has one element from all columns from single band
quorum of size d + h*r - 1

remarks
W = d + hr -1, L = (d + hr-1) / n, R = O(sqrt(n)), afp = 0
number of different quorums is d^n * h * r^(d-1)

6.3.5 f-masking grid quorum system

each quorum contains one column + (f+1) rows of nodes
for 2f + 1 ≤ sqrt(n)

remarks
like multiple T over each others
f-masking, hits lower bound

6.3.6 M-Grid quorum system

each column / row has sqrt(f+1) rows

remarks
L = sqrt(f/n)
f-masking

7 Eventual Consistency & Bitcoin

7.1 definitions

consistency
all nodes agree on current state in system

availability
system is operational and instantly processes requests

partition tolerance
ability to continue operation even while partition exists

quiescent state
no more messages exchanged, shared state consistent

eventual consistency
form of weak agreement, nodes may disagree temporarily
but eventually the system reaches quiescent state

7.2 proofs

CAP theorem
assume two nodes sharing state, cannot communicate
update local state (availability) or not (consistency)

7.3 bitcoin

7.3.1 definitions

bitcoin network
randomly connected overlay network
end users run light, not fully validating, version of nodes
tracks funds of address collaboratively

state of bitcoin
unspent transaction outputs (the UTXO) + global parameters
every node holds complete replica of that state

address
hash(public key), network identifier byte, checksum
stored as base58 which avoids ambiguous characters
20bytes long addresses, impractical to brute force
funds associated is sum of all unspent outputs

output
tuple (amount, spending condition)

spending condition
script, cryptographic puzzle, often singed
can be spend or unspent

input
tuple (reference to output, arguments for spending condition)
reference is tuple (h,i)
h is hash of transaction which created output, i specifies index
the arguments solve the spending condition puzzle

transaction
describes transfer of bitcoins, has inputs / outputs
references outputs are removed, outputs added to the UTXO
maybe less output than input, remainder is called fee
added to memory pool with status unconfirmed
after inclusion into a block remove from pool, set status = confirmed
always spend full amount of coins, add output to self for change
(done because it makes agreeing on transactions easier)

doublespent
multiple transactions spent same output, only one is accepted
nodes do not forward conflicting transactions

bitcoin doubling
as bitcoins traceable, those involved in crime need to be washed
criminal trades his traced bitcoin for less, but clean, bitcoins

proof of work
prove to another party certain amount of resources spent
SHA256(SHA256(c|x)) < 2^244/d in bitcoin network
difficulty adjusted to 10 minutes finding a new block
gives the network time to synchronize

proof of work function
(1) F_d(c,x) is fast to compute with given d,c,x
(2) for fixed d,c finding x is difficult but feasible

genesis block
first, initial block (maybe hardcoded), root of tree

block
contains transactions, reference to previous block, nonce
is broadcast as soon as valid nonce is found
finder of the block imposes chosen transactions to others

reward transaction
first transaction in a block, has dummy input
sum of outputs is fixed subsidy plus all fees

block chain
longest path from genesis block to leaf
only transaction in this chain are valid

monotonic read consistency
if node sees new value, newer reads by same node will always return it

monotonic write consistency
write operations of same node are serialized, executed in order

read-your-write consistency
if node updates value, newer reads by same node will return it

casual relation
causually related operations include
w_a(x) → w_a(y)
r_a(x) → w_a(x)
w_b(x) → r_a(x)
any transitive combinations

casual consistency
if any casual related operations are seen in same order

7.4 smart contract

7.4.1 definitions

smart contract
agreement between two or more parties
blockchain guarantees correct execution (conflict mediator)

timelocked transactions
define earliest time to be included in chain
only released into network after timelock expired

singlesig / multisig transactions
amount of signatures required to claim output

7.5 algorithms

7.5.1 naive ATM

ATM make withdrawal request to bank
waits response from bank
if (OK) dispense else display error

remarks
connection problem may blocks request

7.5.2 partition tolerant ATM

if (bank reachable) then
sync local view with bank view
display error if user balance insufficient, abort
endif
dispense cash, write logs for synchronization

remarks
partition tolerant, no longer consistency guaranteed

7.5.3 node receives new block

add new node to own tree
if (height increased) then
compute UTXO for path until max_node
cleanup memory pool
endif

remarks
switching paths may results in unconfirmed transactions
smart data structures avoid having to recompute everything

7.5.4 parties create 2by2 multisig output

B sends list with inputs to A, A selects own inputs
A creates transaction t_1([I_a, I_b], o = c_a + c_b → (A, B))
A creates timelocked t_2([o], [c_a → A, c_b → B])
A signs t_2 and sends it with t_1 to B

B signs both t_i and sends them back to A
A sings t_1 and broadcasts

remarks
s_1 called setup transaction
s_2 called refund transaction, ensures funds returned
both must be signed by both parties to be valid

7.5.5 micropayment channel, S $\rightarrow$ R, capacity c

create 2by2 multisig (A=R, B=S), c_a + c_b = c
S resigns t_2([o], [c_r $\rightarrow$ R, c_s $\rightarrow$ S]) to buy stuff
at end of period, R publishes last t_2 received from S

remarks
c_r + c_s = c, reduce c_s with amount to pay
only pay fees once, instantly finalized

7.5.6 unlimited micropayment channel

create micropayment channel
but introduce kickoff transaction t_k between t_1 and t_2
t_2 only valid if t_k released into blockchain
t_k needs to be signed at start like t_1
either party can release t_k if wants t_2 to be spend

8 distributed storage

8.1 definitions

diameter
longest distance between any two nodes (using shortest possible)

easy routing
if node does not know result, it should know who to ask

churn
nodes leaving & joining network in time interval (low means little)

topology properties
(1) homogeneous, no single point of failure, dominant nodes
(2) ID assigned from range [1,0), can use decimals
(3) small degree, if possible polyalgorithmic in n
(4) small diameter, easy, predictable & fast routing

8.2 network topologies

fat tree
like a tree, connection capacity equals leave count
only one possible routing path

mesh M(m,d)
like a grid, connects node in x & y direction with others
M(m,1) is a path, M(2,2) a grid with four nodes
routing simple, only flip one bit at a time

torus T(m,d)
like a mesh, but additionally wrap around
T(2,3) is a die, T(3,3) a cube with 9 nodes each side

butterfly BF(d)
d denotes level
constant node degree (d+1), (2(d+1))^d nodes
level 0 is just a point
level 1 adds new line, left/right connected + cross
level 2 adds new line, left/right connected + middle

Cube-Connected-Cycles CCC(3)
degree of 3, replaces corners of hypercube with circles
addressing like (a, b) where b cycle position, a corner position

Shuffle-Exchange SE(d)
d denotes number of bits
connect if differ in last bit
connect if obtained by cyclic right/left shift

skip list
linked list with additional forward links
operations take log n expected time
if inserted promote node if no neighbours are promoted

pancake graph
ID is permutation of 1,...,d
nodes connected if first i numbers of ID are flipped
1234 is neighbour of 3214, 2413 is neighbour of 4213

small world graphs
small diameter but large degree nodes (social networks)

expander graph
sparse graph with good connectivity (clusters are connected)
low degree, small diameter, but hard to route

distributed hash table DHT
implements key-value storage, supporting search, insert, delete
can be implemented with hypercube, using first bits of hash
works with butterfly too, use d+1 layer as replication
to setup, new joining nodes ask authority, use static IP's
recursive lookup builds path to node with object
good for caching, but request amplification & source hidden
iterative lookup builds direct connection to node with object
more expensive logic, but less load on network

set of hash functions
can reuse the same hash function if set needed
repeated hashing hashes k time for the k-th hash function
salted hashing includes the number k into the message to be hashed

8.3 proofs

diameter must be bigger than log(n) / log(d+1) - 2
at most d nodes can be reached, as often as diameter

8.4 algorithms

8.4.1 consistent hashing

hash name of file, hash IP/port of node
store file at closest node determined by hash

remarks
could also store only pointers at designated nodes

8.4.2 DHT

hypercube network with log(n) hypernodes
each hypernode has log(n) nodes
nodes connected to all others in hypernode
nodes connected to core nodes in other hypernodes
some nodes have to change hypernode to balance
if n shrinks/grows to threshold, hypercube is resized

remarks
assume bounded # of join/leaves occur in worst case manner
assume attacker can crash designated nodes at any point
system is never fully repaired, but always fully functional
at any interval, attacker can crash at most log(n) nodes

remarks
each node has log(log(n)) neighbours
nodes are either in core or in periphery of hypernode
log(n) for search/insert, log(n) neighbours with cheating
handles log(n) churn, but not byzantine, privacy, ...

9 Game Theory

9.1 definitions

game
two players, at least two strategies
every possible outcome (strategy profile) has payoff

social optimum (pareto optimal)
if it maximizes sum of payoffs

dominant strategy
if player is never worse by choosing specific strategy

dominant strategy profile
if every player plays a dominant strategy
can only occur in NE's

nash equilibrium NE
player can not improve payoff by unilaterally changing strategy
unilaterally means the other players don't change strategy
games can have multiple nash equilibria
if all players choose dominant strategies

mixed nash equilibrium
if fixed probability for each option is defined this exists

best response
strategy given belief about strategy of others
if best response same strategy for all options, it is dominant

price of anarchy PoA

cost(NE-) / cost(SO), with NE- as NE with smallest payoff

optimistic price of anarchy OPoA
cost(NE+) / cost(SO), with NE+ as NE with highest payoff

mechanism design
focuses on designing games where all behave nicely

auction
each bidder has secret value for good, bids for good
auctioneer sells good to bidder

truthful auction
if no bidder has incentive to bid different price than value

9.2 proofs

OPoA can be O(n)
assume two nodes with connection 1-eps
those two nodes each have n nodes with 0 cost connection
if one caches, cost is 1 + n/2(1-eps) = a
if two cache, cost is 1+1 = 2
the OPoA = a/2 = O(n) for eps to 0, n to inf

Braess' paradox
assume two streets, each with fast then slow part and vice-versa
assume fast street depends on traffic (slower with more traffic)
if connection road build between the two fast parts
then total travel time slower than as if no road would exist

first price auction is not truthful
highest bidder can reduce price by b - eps > b_others

bidding truthful is dominant in second price auction
do case distinction with b_max, b, and value v of item
under/overbidding does not change payoff, or decreases

NE for selfish cashing can be done
as mechanism designer can choose payoffs
can payout if no one caches, or vice versa for all subset

tit-for-tat
always mirror the action of the other player
enforce with cryptocurrency, reputation systems
give something for free to bootstrap

9.3 games

prisoners dilemma
if both defect get high penalty
if both cooperate get low penalty
defector get no penalty if the other cooperates
if repeated, called iterated prisoners dilemma

selfish caching
each node has demand d for file
each node can cache file locally for cost 1, or ask another node
if asking other node, there is a cost d for transfer
if no one caches, cost is +inf

rock-paper-scissors
best response changes with each move, therefore no pure NE
expected payoff 0 with 1/3 p mixed NE

bidding with full payout
each bidder has to pay in full, even if he does not win
will go indefinitely because rationale is to keep bidding
either don't play, split earnings (collude), or start highest

9.4 algorithms

first price auction
sell price to highest bidder

second price auction
sell price to highest bidder for price of second highest
use concept for selfish cashing

selfish caching
assume storage cost 1, transfer from node n costs d_n
choose node n with highest demand, it needs to cache
remove all nodes from set of nodes S with d_n < 1
repeat till S empty

9.5 exercises

pure nash equilibrium NE
write down for each entity best response strategy
find assignments which are allowed by strategies

social optimum SO
choose assignment which would be best for all
could also be a NE

price of anarchy PoA
take worst NE, then NE / SO, should be ≥ 1

optimal price of anarchy OPoA
take best NE, then NE / SO, should be ≥ 1

mixed nash equilibrium MNE
define p,q probabilities for player 1,2 to play option 1
then (1-p), (1-q) defines probability for player 1,2 to play option 2
write down utility for each player depending on p,q
p(payoff_for_q ∗ q + payoff_for_not_q ∗ (1-q)) + reverse
then check where payoff = 0 for both players, this is MNE
proof that it is the only one

10 Locking

10.1 general

focus
multiple faithful processes with shared memory
practical performance the most important factor

wait by blocking
let scheduler start a new thread, and pause

wait by spinning, busy-trying
keep trying to unlock by spinning on memory location
but memory changes not instant (could use memory barrier)
but memory does not guarantee sequential memory
but processors reorder instructions

10.2 testAndSet

java api
single binary value
stores true and returns value from before
getAndSet(true) is same
provokes a lot of bus traffic

TTAS
read fields repeatedly and wait till correct value read out
then use testAndSet to finalize operation
avoids some unnecessary bus traffic
but on testAndSet, invalidation storm happens at all cores

contention
if multiple threads try to acquire lock at the same time
low if few, high if many

exponential backoff
wait for random time, each try double possible range

read-write lock
single shared integer, set to -1 for write, set to >0 for read
readers can starve writers

ticketing lock
single shared integer, first part is head, second part is tail
on locking, increment tail value, set as current ticket number
on waiting, wait till head equals current ticket number
in release, increment head by one

10.3 queue locks

problems tackled
less cache-coherence traffic (no spinning on same location)
better critical section utilizing (instant control switch)

generally
enqueue thread, each spins on different location
each new thread is notified by his predecessor
first-come first-serve fairness

array-based
boolean array, entry is true if respective thread allowed
add pads between the entries to avoid false sharing of cache line

CLH queue lock
list of nodes, each node represents thread
each node has field is_waiting (boolean), pred (predecessor node)
on lock acquire, set as tail of list, set pred, set is_waiting to true

on waiting, spin on pred.is_waiting, wait for it to be false
on unlock, set this.is_waiting to false
can reuse pred node for future accesses
spinning on different core could be an issue

MCS queue lock
list of nodes, each node represents thread
each node has field is_waiting (boolean), next (successor node)
on lock acquire, set tail to self, set (old tail).next to self
on waiting, set is_waiting to true, spin till it is set false
on lock release, set next.is_waiting to false
spins on same core

queue lock with timeouts
allow timeout (max time caller wants to wait)
trivial with backoff algorithms, difficult with queues
do not remove nodes, rather mark node as abandoned
do so by setting itself as value of pred of next node
waiting node spins on in field pred
if (field is null) then keep waiting
elseif (field is static node AVAILABLE) then proceed to critical
else repeat on pred.pred ("change node")

11 concurrency

11.1 definitions

coarse-grained synchronization
acquire lock for whole data structure

fine-grained synchronization
split the object into independently synchronized components

optimistic synchronization
search with no locks, verify after talking locks

lazy synchronization
postphone work (split logical, physical removal)

non-blocking synchronization
use of atomic operations such as compareAndSet

11.2 concurrent list

11.2.1 concurrent reasoning

define invariants, show they hold after creation
show that no thread can make property false

11.2.2 freedom from interference

only methods where concurrent reasoning holds can access

11.2.3 abstract vs concrete

abstract value defines the perceived functionality
concrete representation defines the implementation

11.2.4 invariants

sentinels (head, tail) are not added/removed
keys are unique and in order

11.2.5 safety property

linearizability (identify linearization point, atomic step)

11.2.6 non-blocking properties (liveness)

wait-free, every call finishes in finite number of steps
lock-free, if some finish in finite number of steps

11.2.7 implementations

coarse-grained synchronization
every thread must acquire single lock from list
linearization point is where the lock is aquired
satisfies same progress property as lock

fine-grained synchronization
hand-over-hand locking, always pred & curr is locked
all threads must acquire in same order to guarantee progress
starvation free because of lock and no deadlock possible

optimistic synchronization
search without acquiring locks, then lock & check
needs to traverse a second time in validate
not starvation-free

lazy synchronization
add boolean to node called marked
contains() wait-free because can simply traverse list
add() locks, checks for not marked, then adds
remove() lazy, sets marked to true, then changes pointers
add()/remove() not starvation free
linearization on setting the mark

non-blocking synchronization
cannot use compareAndSet (two following nodes removal)
use other api which includes marked bit in reference
needs to take clean-up approach to avoid 2nd traverse of list
remove() marks elements as removed, then tries once to remove
add()/remove() lock-free, contains() wait-free
add()/remove() look for marked, clean up & restart if found

11.3 concurrent hashing

11.3.1 modes

open addressing (hash refers to single item)
closed addressing (hash refers to bucket of items)

11.3.2 properties

easy to parallelize (disjoint-access parallel)
deal with collisions (different items, same hash)

11.3.3 approach

define base hash class with contains, add method
abstract acquire (before contains, add)
abstract release (after contains, add)
abstract policy (checks if resize should be called, after add)
abstract resize (after policy returned true)

11.3.4 implementations

course grained hash set
acquire(), release() use simple lock
policy() checks for 4 average, resize() doubles buckets

striped hash set
each bucket has own lock, lock array size not changed
resize() acquires all locks in specified order
each lock is responsible for all buckets b mod n

refined hash set
introduce reference with owner, boolean
owner can resize locks, boolean indicates if resize in progress
acquire() needs to check after locking if the locks still valid
resize() sets itself as owner, acquires all locks, then resizes

12 Consistency & Transactional memory

12.1 concurrent objects

sequential object method specification
precondition (state which has to hold)
postcondition (return values, exceptions)
side effects (what happens when calling)
must assume meaningful state only between calls of method

keywords
program order (method execution order of single thread)
compositional (system fulfils if all objects fulfil)
execution in isolation (no other threads take steps)

progress conditions
non-blocking (any pending invocation has correct response)
blocking (one thread can deny others progress)
wait-free (invocation finishes in finite number of steps)
bounded wait-free (like wait-free, but bounded steps)
population-oblivious (performance not dependent on thread count)
lock-free (some invocations are wait-free)

dependent progress conditions
progress occurs if lower layer provides some guarantees
lock-based algorithms can only ever guarantee dependent progress
deadlock-free (some process will eventually make progress)
starvation-free (any process will eventually make progress)
obstruction-free (any isolated process finishes in finite steps)

method calls should appear
(1) sequentially one-at-a-time
(2) effective in real-time order if separated by quiescence
(3) effective in program order
(4) effective instantaneously at some moment (linearization point)

linearization point
where method effect is visible to all other users

if locks, the critical section
if no locks, reference / boolean writes or similar

quiescent
if no pending method calls (”break”)

quiescent consistency
needs (1), (2)
can reorder any calls in groups separated by break
w_a(0) break; w_a(1), r_a(0), r_b(1)
is non-blocking, compositional
use in printer

sequential consistency
needs (1), (3)
can shift around calls but no reordering
w_a(1), w_b(2) break; r_a(1), r_b(2)
there may be one than one valid orders possible
processor provides memory barriers to deny reordering
is non-blocking, not compositional
use for banking

linearizability
is composable
needs (4)
is non-blocking, compositional
use for stock-trading

12.2 transactional memory

why locking is bad
priority inversion (low thread holds lock for high)
convoying (thread holding lock is descheduled)
deadlock (threads lock same resources in different orders)
maintenance relays on conventions, comments

why compareAndSet is bad
operate only on single word, algorithms get complex
can't implement multiple-compareAndSet

why compositionality is bad
can not compose multiple atomic calls to single atomic
modularity of algorithms hard

transaction
if (no error during execution) then commit else aborts
are serializable (coarse-grained linearizability)
no dead-lock, no live-lock
executed speculatively, makes tentative changes
can be nested (modularity), parent may not abort if child does
globally ”locks” with all other active atomic sections

12.3 software transactional memory (STM)

12.3.1 transactional thread

has onCommit, onAbort, onValidate callables
create thread for each transaction which runs callable
retry transaction if no error till status is committed

12.3.2 transaction

has status active, aborted, committed
exposes this status to the contention manager

12.3.3 atomic objects

way for communication between transactions
needs get/set for all fields, atomic arrays
needs copyTo method for the different transactions
openRead / openWrite for getter/setter, afterwards call validate()
validate() returns false if current transaction has been aborted

12.3.4 contention manager

resolves conflicts between two transactions
can abort transaction by getAndSet to status
backoff (back-off doubling each time, if limit reached abort)
priority (priority by timestamp, abort younger, else wait)
greedy (priority by timestamp, abort younger & waiting)
karma (priority by work done, abort less work)

12.3.5 obstruction-free atomic object

setup
each object has tree fields (owner, old version, new version)
owner is last transaction which accessed the object
old version is object state before transaction arrived
new version has changes from transaction if any

determining current version
if (owner committed) then new version is current
if (owner aborted) then old version is current
if (owner active) then no current version

field access
if (owner committed) then set old = new, sets new
if (owner aborted) set new = old, sets new
if (owner active) asks contention manager

12.3.6 lock-based atomic object

concept
remove need for double reference traversal
instead lock on writing (reading doesn't need to)
global version clock, increased each time a transaction commits

object fields
stamp (clock of last time written to)
version (instance of sequential object)
lock (is a lock)

transaction
uses local read/write set of objects (with versions obtained)
does all changes to this set

validate of transaction
locks all write objects, compareAndSet to increment global clock
check all read objects are not locked, validates local read versions
updates all stamps, versions of write set objects

12.4 hardware transactional memory

cache coherence
detects synchronization conflicts
listens on the bus (snooping) to detect changes

MESI
modified (line is modified, needs to be written back)
exclusive (line not modified, and not on other cores)
shared (line not modified, but on other cores)
invalid (line invalid)

MESI load actions
if (exclusive) then write back modified, invalidate shared
if (shared) then change exclusive to shared

transaction extension
add transactional bit to cache line tag
if (bit set && modified) then do not write back
if (bit set && invalidate) then abort transaction
clear bit on committing transaction

limitations
aborts can be hardware (don't retry) or conflicts (retry)
size of transaction is limited to cache size
length limited to scheduling by OS / next cache clean-up
not fully associative cache may has impossible transactions
transactions can starve each others

13 blockchain / coins

13.1 piChain

unite block-chain (simple, fault tolerant, but eventual consistency)
with consensus protocols (strong consistency, but poor scalability)

target properties
fault tolerant (no byzantine, but crashes)
fast (no base overhead, strong consistency within one RTT)
quiet (no heartbeat needed, no work no messages)
scalable (no quadratic number of messages)
light (no explicit leader, simple architecture)
available & consistent (only short united phases needed for consistency)

transactions
executable commands
unique id (node which created transaction, sequence number)
broadcasted to whole network
every transaction will eventually be in a block

blocks
many or single transaction
unique id (node which created block, sequence number)
depth field (number of transactions including ancestors)
contains pointer to parent block (deepest block creator has seen)

node states
node is either quick, medium, slow
when quick, creates block as soon as new transaction found
when medium, wait with creating block longer than quick (wait 2t)
when slow, only create block if fast nodes somehow don't (wait 4t)
node promotes when new block created
nodes set to slow when new block seen && (deepest || creator is quick)

strong consistency
nodes can only commit block if parents are transitively committed
to commit a block, node must convince majority twice with paxos

comments on approach
can commit few times to shorten chain, or often for fast consistency

paxos adaptation
node communicates with other nodes, local storage contains only blocks
T_store called b_supp, T called b_prop, b_max called T_max
b_new is the new block, b_com is c
can skip phase 1 by directly sending propose(b_new, empty)

healthy system
if only one quick node, then paxos can run in parallel
if all slow nodes, few may become medium, but only one quick

13.2 ethereum

turing-complete scripting language to serve as base for programs
value-awareness (can withdraw fine tuned amount)
stateful (scripts can have state)
blockchain-awareness (scripts know where they are in chain)

state (accounts)
20-byte address, is target & source of transactions
nonce (counter to make sure transaction only processed once)
ether balance, contract code (if any), storage (default is empty)
can be externally owned (private keys) and contract owned
all forms of addresses have same possibilities

messages
like transactions, but with storage
if (target is contract) then can have return value

transaction
singed by externally owned account
contains recipient, signature, ether & date to be sent
STARTGAS (max amount of steps) & GASPRICE (fee per step)
execution terminates if STARTGAS empty, else rest is transferred back

state transition
validate transaction (signature, nonce, balance of sender)
subtract GASPRICE ∗ STARTGAS, increment nonce from sender account
init GAS = STARTGAS, subtract some GAS to pay for transaction
locate receiver, transfer ether, if contract, run its code
if transfer failed because GAS = 0, then revert state changes
add fees to miner account, refund rest to sender

code execution
stack-based ethereum virtual machine code
can be written in Serpent (high level language)
32byte stack, memory byte array, key-value storage of contract

blockchain
each block stores transaction list and most recent state
check transactions, previous block, proof of work are valid
can efficiently store the state with pointers

applications
financial (contracts, wallets, payment)
token systems (property, coupons, coins)

possible implementations
namecoin (DNS like, persist name to storage of contract)
file storage (continuously pay out anyone holding a chosen file block)
organisation (execute sub-contracts as agreed on)
saving wallet (bank which can pay out 1% as security)
crop insurance (connect with weather feed, pay out when bad)
data feed (pay out all which are within 25-75% of median)
escrow (signature with different weights)
cloud computing (randomly pay out if participant is correct)
gambling (difference on block hash)

14 Exercises

14.1 asynchronous riddle

group of people each continuously enter room with switch
find out when everybody visited the room at least once

switch position known
all turn switch on exactly once
selected leader turns switch off, counts to n-1

switch position unknown
all turn switch on twice
selected leader turns switch off, counts to 2n-2