Distributed computing: Quick reference

vishvAs vAsuki

March 30, 2012

Contents

1 Themes

Coordinating and orchestrating a bunch of processes. There is no tight coordination amongst processes. Coming to an agreement.
Usually don't care about efficiency: just try to prove safety and correctness.

1.1 Characterization of research effort

1.1.1 Safety and correctness: Use of formal methods

Makes heavy use of formal methods of proving program correctness. With formalism, proofs tend to be shorter and more elegant than 'intuitive' proofs. Once you axiomatize the system well, you don't make much use of intuition. You make heavy use of theorems that are not intuitive.

1.1.2 Efficiency: Implementation, Engineering

Design and implimentation of good API's.

2 Reasoning about distributed computation

2.1 Common assumptions

A channel eventually (maybe at $t = \infty$) delivers all packets without corruption or dropping, in the correct order.
Processes can run at arbitrary, non 0 speed.
Don't care about efficiency, focus on safety and progress.
Assume you have infinite buffers.

2.1.1 Synchronized communication

Sometimes, this is needed. Max time to deliver a message is some t.

2.2 Proving algorithm efficacy

Axiomatize the system, and use pure logic.
Usually 2 stages: safety property: you won't fall into a wrong state; and progress property.

2.2.1 Importance and need

Distributed computation is very hard to debug: bugs are hard to reproduce, very scary for usual programmers; so, proving it correct is essential.
When you have the right theory, a formal proof is much shorter than an intuitive proof.
Also, English sentences are often ambiguous.

2.2.2 Common tricks

Decompose computation into processes which are independent of network structure.
Use induction.
Often, need to break cycles in the network, to avoid deadlock, livelock. Then, induction is possible.

2.3 Properties of programs

It is easier to reason about properties of a program, rather than its the program text.
'Interplay between safety and progress properties is design.'

2.3.1 Safety of the computation: problems

Aka rightness property. Claim: The computation will do no harm.
While shared memory may be common in parallel programming, it is uncommon in distributed computing. Shared memory and message passing are two common parallel programming paradigms, of which the latter is necessarily safer.

2.3.1.1 Race conditions: Readers/ writers problem

If memory locations are shared between threads, there can be race conditions. Usually, many readers can read simultaneously, but only when writer is using the resource, no readers or writers should be allowed: mutual exclusion/ mutex.

2.3.2 Progress of the computation: Problems

Aka liveness property. Claim: The computation will eventually produce the correct answer.
Safety proofs can be usually done without using progress properties, but not vice versa. Eg: to prove x increases, prove $tr(x = m) \wedge stable(x \geq 5)$.

2.3.2.1 Deadlock

A is waiting for a resource B is holding, B is waiting for a resource A is holding; neither will release until it acquires both processes.
Solution: Either work under the assumption that this never happens like most OS's, or prevent, detect and break it. So, there is no progress.

Expressed as a safety property $\forall i$ stable waiting(i).

2.3.2.2 Livelock

Eg: A asks B if it, or anyone it can reach has x; B doesn't have it, but asks all nodes connected to it, including A, exactly the same question. If A or B do not break the cycle, there is no progress.

2.3.3 Maximality

Is this program the most efficient? Or requiring the least communication?

2.4 Action systems

Aka State Transition system.

2.4.1 Motivation

Highly abstract setup to reason about distributed computing. Many different looking algorithms for the same problem : eg message passing vs shared memory vs token ring based etc.. turn out to boil down to the same logic in an action system.
It is easier to reason about properties of a program, rather than its implementation.
Can 'refine' an action system in various ways to get actual implementations of the algorithms.

2.4.2 Definition

2.4.2.1 Objects

A program is collection of objects. Every object is a collection of states and actions. If there is communication, it is with shared variables. Usually only one object is the program.

2.4.2.2 States

Specified by some variables. Initial conditions.

2.4.2.3 Transitions or Actions

When in state s, where can you go next?; visualize as a directed graph among states. $p \to x = 5$: p is the guard; x=5 is the command. If guard is absent, 'true' is assumed to be the guard. If guard is true, the command is 'active' else it is inactive.
Always, 'skip' or 'noop' command is included.
All actions terminate.

2.4.2.4 Execution rule

Minimal progress Until all guards are false: arbitrary non-skip action whose action is true is executed. So not fair!

Weak fairness Each action is executed infinitely often; Can think of even inactive actions being executed: they just have no effect.
If the guard of an action remains true continuously, it is eventually executed effectively.

Strong fairness If guard of the action is true infinitely often, then action is executed effectively infinitely often.
So, in a certain recurring state, if k actions are active, they all are executed effectively infinitely often. In weak fairness, they might have been executed when inactive.

Termination No 'termination state' specified: theoretically, 'program' runs for ever; detecting termination is an implementation issue. It happens when all actions are ineffective. Eg: detecting fixed points.

2.4.3 Reasoning about properties

2.4.3.1 Properties vs predicates, connectives

Safety and progress 'Properties' of a program are defined to be distinct from predicates. Predicates related program variables with logical operators, Eg: $p \wedge r$ vs p *co* q. Quantification implicit p *co* $q \equiv \forall x, S :: \{p\}\, S\, \{q\}$.
But, $\wedge$, $\Longrightarrow$, $\equiv$ still used with properties to indicate 'both hold' or can be inferred from etc.. But, they are used purely amongst properties or purely among predicates. property $\Longrightarrow$ predicate: like $p \vee (p$ *co* $q)$ illegal.
Quantification: $(p$ *co* $q) \wedge (stableq)$ indicates $(\forall x :: p$ *co* $q) \wedge (\forall x :: stableq)$.
$p \Longrightarrow q$ is not a property; but $invariant(p \Longrightarrow q)$ is.

2.4.3.2 Auxiliary vs program variables

Program variables are variables used in the action system text. Auxiliary variables are extra variables used to reason about the action system, which depend only on the history of program variables. Eg: mx = Max value x has achieved so far.

2.5 Safety properties of Action systems

Usually, use intuition to identify some target property of the action system, express it using co operators, identify the initial conditions, basic properties of the system, thence derive the target property.

2.5.1 The constrains/ co operator

A type of temporal implication. p *co* $q \equiv \forall t : \{p\}\, t\, \{q\}$. MUST Show $\forall(g \rightarrow s) : \{p \wedge g\}\, s\, \{q\}$. If cannot show, the claimed property is false!
So, once p holds, q continues to hold until $\neg p \wedge q$ holds.

2.5.1.1 Properties of co

Many properties follow from implication: See inference/ propositional logic ref; F co p. p co T. Can strengthen LHS or weaken RHS; it is transitive. If $(p$ *co* $q) \wedge (r$ *co* $s)$, Can do this: $(p \wedge r)$ *co* $(s \wedge q)$ or $(p \vee r)$ *co* $(s \vee q)$.
Any action system includes skip statement; so if p *co* q, $p \Longrightarrow q$ should hold.

2.5.1.2 Formalizing safety properties

Take the statement you want to write using proper notation. Set initial state p. After any action, including skip, how can the state possibly change? : list

these as $\{q_i\}$, join them thus: $p\ co\ \bigvee_i q_i$; then simplify. Note q_i must specify the state fully: else error.
Eg: d does not change as long as c remains true. $d = m \wedge c\ co\ (c \wedge d = m) \vee (d \neq m \wedge \neg c) \vee (d = m \wedge \neg c) \implies d = m \vee \neg c$.

2.5.1.3 Elimination theorem

If $x = m\ co\ q$ and p does not have any free variables other than x: $p\ co\ \exists m :: p[x := m] \wedge q$.
Very useful in eliminating free variables. Eg: given $x = m\ co\ q$, show stable p.

2.5.2 Stable predicates

Stable predicate is a set of states outside which no transition can ferry the program. Visualize using a set of states.
stable p := p co p.

2.5.3 Fixed point x

Once you hit state or state set x, you never get out. Indicates termination.
Holds in any state where all actions are ineffective. So, to compute fixed point predicate, take $\wedge(\neg guard_i)$ where i ranges over all actions.

2.5.4 Invariant p

True initially, and stable. So, show: init conditions $\implies$ p; $\forall(g \rightarrow s) : \{p \wedge g\}\, s\, \{p\}$.

2.5.4.1 Substitution axiom

While reasoning about a program, can replace all occurances of an invariant with T; or vice versa.

2.6 Progress properties of action systems

Action system: $\{g_i \rightarrow S_i\}$

2.6.1 transient p

tr(p): p guaranteed to be falsified by a single action. $tr(p) \implies tr(p \wedge q)$.

2.6.1.1 Defn Under weak fairness exec rule

$p \implies ((\exists i :: g_i) \wedge (\forall i :: \{p \wedge g_i\}\, S_i\, \{\neg p\}))$. If you can't show this, p is not transient.

2.6.1.2 Defn Under minimal progress exec rule

$\exists i :: \{p\}\, g_i \rightarrow S_i\, \{\neg p\}$; ie $\exists i :: (p \implies g_i) \wedge (\{p\}\, S_i\, \{\neg p\})$.

2.6.2 p ensures q

If p holds, it continues to hold until q becomes true, and q will become true. Combines both safety and progress properties.
$(p\ en\ q) \equiv ((p \wedge \neg q)\ co\ (p \vee q)) \wedge tr(p)$.

2.6.3 Temporal logic

$p \mapsto q$: if p holds, q holds now or eventually. If $p \implies q$, $p \mapsto q$. $p\ en\ q \implies p \mapsto q$.
[**Incomplete**]

2.7 Ordering events in distributed computation

Processes cannot be perfectly synchronized in their evaluation of 'time': clocks drift.

2.7.1 An irreflexive partial ordering

Independent events cannot be ordered without ambiguity: only causality defines 'order': If $a \prec b$, a can affect b.
Events in a single process are completely ordered. There is no simultaneity. A process is a sequence of events.
For any message, send event preceeds receive event.
$\prec$ is transitive.
Concurrent events: $a \not\prec b, b \not\prec a$: can't causally affect each other.

2.7.1.1 Impossible orderings

Eg: A happens before B, B happens before A. Can't go back in time and alter events to alter chances of your birth.
Apprears as cycles in time-line graphs. These can creep in when a long sequence of distributed computing, involved sends and receives are involved.

2.7.2 Time-line graphs

Take vertical line for each process. y axis is time. Directed edges between points in various processes show msg send and recieve events.
Observe causality paths going vertically, diagonally in this graph. They illustrate the ordering to be obeyed by the logical time assigned to each event.
Easiy to identify independent events: points along multi-edges between 2 points in the causality graphs.

2.7.3 Assign numbers to events: Logical clocks

Contrast with physical clocks. C_i: clock for process P_i. Global clock $C(x) = C_i(x)$ if x happens in P_i.

2.7.3.1 Clock condition

$a \prec b \implies C(a) < C(b)$.

2.7.3.2 Implementation rule

P_i increments C_i between 2 events.
When a message m is sent $P_i \to P_j$, the $C_i(send(m))$ is sent along. Then, $C_j(receive(m)) = \max(C_i(send(m)), C_j) + 1$.

2.7.3.3 Imposing total order

Just break ties arbitrarily.

2.8 Common knowledge

k(A,p): A knows p. $k(A, p) \implies k(A, k(A, p))$.

2.8.1 Coordinating attack problem

Aka attacking generals problem. 2 generals on top of hills, enemy in the valley in between. If attack executed simultaneously, it succeeds; else it fails. Generals communicate only by sending messengers. No general will attack if he is not sure that the other general will also attack.
So, no attack can be arranged if it was not prearranged. No pre-arranged attack can be called off.

3 Designing distributed computation

3.1 By designing action systems

By specifying the safety and progress properties the distributed computation should have, you can then design a compliant action system. Then you can refine this in various ways to get different looking implementations.

3.2 Coordinating processes

There is always an initiator process.

3.2.1 General signalling constructs

See programming ref. Callback, polling.

3.2.2 Synchronization

3.2.2.1 Blocking

Process A may be listening in the channel, and may be blocked until it receives a message.

3.2.3 Resource allocation

Resource conflicts often arise: resolving them involves giving one process preferential treatment over others.
Important progress conditions: avoid deadlocks, resource starvation.
Fairness: No resource starvation for any process. The process which is chosen to get a resource during a conflict is not always the same. You can't have this without randomization. There should be randomization either in the initial conditions, or in resolving conflicts.

3.2.3.1 Resource starvation

Use timers and priorities.

3.2.3.2 The dining philosophers problem

Models situation where each conflictable resource is shared between no more than 2 processes. So, you got a undirected 'conflict' graph among processes. A process has states: processing/ thinking, hungry/ waiting for a resource, using the rosource/ eating, with possible transitions from one to the next. A hungry process eats when it has all the forks.
Safety condition: Two neighbors cannot eat simultaneously.
Fairness condition: every hungry process gets to eat.

Starting from random initial condition (Chandy, Misra). Overlay some random directed acyclic priority graph among processes. The node at lower level has higher priority access to the shared resource, when there is a conflict. Upon eating, a process always turns the edges inwards/ acceeds priority to neighbors. This ensures fairness. Initially, lower priority process controls the resource.

Drinking philosophers problem A generalization of dining philosophers problem: neighbors can drink together, as long as they drink from different bottles.

3.2.3.3 Semaphores for mutual exclusion

Associated with a resource. Binary semaphore aka lock. A process gets access to resource by 'acquiring' / holding semaphore by executing 'P op', releases it with V op.

Weak semaphore guarantees mutual exclusion but not absense of starvation. Strong semaphore guarantees both.
Usually implemented as a process.

3.2.4 Call-back

Processes requesting access register with a coordinator process, who will determine when the resource is available, and call back.

3.2.5 Randezvous

Processes reader and writer. sender must send exactly when reader is ready to receive: unlike usual case where receiver is blocked in channel.get() operation until sender sends.

3.3 Distributed program paradigms

3.3.1 Central coordination

3.3.1.1 Master slave architecture

You have one master, many slaves to do the actual work.

3.3.1.2 Using coordinator

In master slave architecture, the master is the weakest point in the system. Instead, can replace master with a coordinator, which then chooses one among a set of masters. But, now the coordinator is the weakest point in the system.

3.3.2 Layered computation

Many different layers of superposed distributed computations happen. The layer above only can look at the state of processes in the layer immediately below.
Eg: can superpose a layer above superposed computation which takes snapshot in order to collate the states at a single process.

3.3.3 Recursive process networks

A process calling internally spawning a distributed computation which includes a process like itself.

3.3.3.1 Factorial finding

Finding factorial of numbers which come in a channel. Visualize process diagram.

```
def processFactorial(channelIn,channelOut,0) = 1
def processFactorial(channelIn,channelOut,n) =
var channelInOut = Buffer()
var channelInFact = Buffer()
var channelFactOut = Buffer()

def processIn(channelIn,channelInFact,channelFactOut) =
channelIn.get()>n>channelInOut.put(n)
|processFactorial(channelInFact,channelFactOut)
|processIn(channelIn)

def processOut(channelOut,channelInOut,channelFactOut) =
channelInOut.get()
>n>if(n == 0) then channelOut.put(1)>>stop() else
(channelFactorialOut.get()>m>channelOut.put(m*n))
>>processOut(channelOut,channelInOut,channelFactOut)

processIn()
|processOut()
|processFactOut(channelInFact,channelFactOut,n-1)
```

3.3.4 Discovering a spanning tree for process network.

Network structure initially unknown. An initiator node is specified as the root node.
Useful in breaking cycles: Needed to avoid deadlocks and livelocks. Useful whenever a process needs to gain information by polling all processes. Eg: Querying a distributed database.
Each process does this: def process (queryId, 'will you join the tree as my child?') = if it has not committed to being another process's child, return yes; pass on similar query to other neighbors.
The tree is maintained at each process: by each process knowing its parent, and/ or by knowing its children.

3.4 Diffusing computation

3.4.1 'Underlying' computation

Many processes. There is a (usually undirected) graph over these, indicating bidirectional communication channels. There is an initiator process. Initiator is initially the only process active in the underlying computation. By sending messages, an already active process may make child processes active. Active processes may turn 'idle', as far as underlying computation is concerned.

3.4.2 Termination detection

Initiator process must be able to tell whether the underlying computation is terminated. Need superposed computation to see if all processes are idle, all channels are empty.

3.4.2.1 Solution with acknowledgements

Require all messages to be ack-ed: thence see if channel is empty. Each node maintains a variable indicating if it sent messages yet to be acknowledged: msgs sent - acks received. Maintain a tree of nodes which are either active, or have uncknowledged messages, by having each node know its parent. Each node sends an ack to its parent only when it is idle and it has all its messages have been acked: that is, when it is a leaf node. When initiator gets all the acks it expects, it knows that the tree is empty but for itself.

3.4.2.2 Polling for state

Periodically poll all processes (having first discovered a spanning tree) to check if they are idle, received acks for all messages.

3.4.2.3 The uni-directional ring: without acks

Acks suck bandwidth. Instead, periodically, all processes record the number of messages sent, received. A superposed computation checks all processes periodically to see if the channels are empty, the outgoing channel is empty.

3.5 Global snapshot

3.5.1 Application

3.5.1.1 Checkpointing

Save the state for all processes and channels. Processes may crash: want to recover easily.

3.5.1.2 Census

How many cows are there in a large ranch? Prople in USA?

3.5.1.3 Consistant state

If any event is included, all preceding events should be included.

3.5.2 Algorithm

Processes are in 2 states: 'snapshot done'/ red, 'snapshot not done'/ white. All messages carry a color: set by the sending process. Every process turns red

just before receiving a red message, or any time before that; 'redness' is stable. So a white process recieves only white processes. White messages received by a red process are recorded as state of the channel.

3.5.2.1 Proof of correctness

There exists an logical ordering where all white events precede red events. Proof intuition: Take any logical ordering of the events; observe: if a red event precedes a white event, you can interchange them; can thence decrease the number of red events before the last white event.

3.6 Fault tolerance

3.6.1 Motivation

Some processes work perfectly, as they are supposed to. Others fail all the time. Diabolical ones fail inconsistently, or adversarially: can say one thing to one process, and another thing to another. Same with circuits: 'stuck at 0' in EE.

3.6.2 Taking majority

Have k identical redundant circuits. Output the majority.
This is not as good as getting a consensus. [**Find proof**]

3.6.3 Consensus

3.6.3.1 Byzantine generals problem

There is 1 general (process) and many soldiers. Generals and soldiers may be good or bad. A bad general can tell some soldiers 1 or other soldiers 0. All good soldiers must mutually agree on a common value. If the general is good, good soldiers must agree on the general's initial message.
Isomorphic to consensus problem: the general is the setting of initial values in the consensus problem!

3.6.3.2 Consensus problem

There are g good processes and b bad processes. If all have same initial value, consensus value is m for good processes. Even otherwise, good processes must agree on a common value.
Solvable iff $g > 2b$ and communication is synchronous.
Often algorithms described in terms of 'rounds'.

Lower bounds For deterministic alg: Ye need at least b+1 rounds.

Every good process can't just take majority of values it receives: a bad process may send different values to different good processes, causing them to settle on different values.

Upper bounds Simple alg: in b+1 rounds, transmit $n^{n/3}$ bits.
But, there are polynomial time algs which work in b+1 rounds. Maybe use digital signatures.
Randomized algorithms: do it in 2 rounds.

4 Problems

4.1 Agreement on public randomness

n players with n random coin-tosses; communicate by broadcasting. t bad players broadcast last. How to calculate public coin toss while tolerating mischief of bad players?