High-Level Executable Specifications of Distributed Algorithms

Y. Annie Liu

Computer Science Department
State University of New York at Stony Brook

joint work with
Scott Stoller and Bo Lin

Specification of distributed algorithms

distributed algorithms are at the core of distributed systems.

understanding them and proving correctness remain challenging.

specification of distributed algorithms:

- pseudocode, English: high-level but lacking precise semantics
- formal specification languages: precise but often lower-level
- high-level programming languages: not sufficiently high-level but precise and executable

e.g., distributed consensus: Paxos, simple to full, much to study

This work: high-level executable specifications of distributed algorithms

use a simple and powerful language, DistAlgo: very high-level

- distributed processes as objects, sending messages
- yield points for control flow, handling of received messages

\+ await and synchronization conditions as queries of msg history

- high-level constructs for system configuration

exploit high-level abstractions of computation and control

1. high-level synchronization with explicit wait on received msgs
2. high-level assertions for when to send msgs and take actions
3. high-level queries for what to send in msgs to whom
4. collective send-actions for overall computation and control

experiment with important distributed algorithms

- including Paxos and multi-Paxos for distributed consensus
- discovered improvements to some, for correctness & efficiency

Not discussed in this paper

compilation, optimization to generate efficient implementations

transform expensive synchronization conditions

into efficient handlers as messages are sent and received,

by incrementalizing queries, especially logic quantifications,

via incremental aggregate ops on appropriate auxiliary values

use of message history $\longrightarrow$ use of auxiliary values

[Liu et al OOPSLA 2012] and much prior work

DistAlgo: distributed procs, sending msgs

process definition

`class P extends Process: class_body` with `run`

defines class `P` of process objects, with private fields

process creation

`new P(...,s)` `newprocesses(n,P)`

creates a new proc of class `P` on site `s`, returns the proc

sending messages

`send m to p` `send m to ps`

sends message `m` to process `p`

usually tuples or objects for messages;

first component or class indicates the kind of the message

DistAlgo: control flows, receiving msgs

label for yield point

```
-- l
```

defines program point `l` where the control flow can yield to handling of certain messages and resume afterwards

handling messages received

```
receive m from p at l:  stmt                receive ms at ls
```

allows handling of message `m` at label `l`; default is at all labels

synchronization

```
await bexp: stmt  or ... or timeout t: stmt
```

awaits value of `bexp` to be true, or `time` seconds have passed

high-level queries of sequences of messages `received` and `sent`
including quantifications, both existential and universal

DistAlgo: configurations

channel types

```
use fifo_channel
```

default channel is not FIFO or reliable.

message handling

```
use handling_all
```

all matching received msgs not yet handled must be handled at each yield point. this is the default.

logical clocks

```
use Lamport_clock
```

call `Lamport_clock()` to get value of clock

1. Explicit wait for high-level synchronization

synchronization is at the core of distributed algorithms:

wait for conditions to become true before appropriate actions;

need to test truth value of conditions as msgs are received

principles:

1. specify waiting on conditions explicitly using await-statements
2. express the conditions using queries over `received` and `sent`
3. minimize local updates in actions

example: commander in multi-Paxos:

- spawed by a leader for each adopted (`ballot_num`, `slot_num`, `prop`)
- try having it accepted by acceptors & send replicas the decision
- in case preempted by a different ballot num, notify the leader

Example: Commander in multi-Paxos [vR11]

process $Commander(\lambda, acceptors, replicas, \langle b, s, p\rangle)$
  var $waitfor := acceptors$;

  $\forall \alpha \in acceptors : send(\alpha, \langle \mathbf{p2a}, self(), \langle b, s, p\rangle\rangle)$;
  for ever
    switch $receive()$
      case $\langle \mathbf{p2b}, \alpha, b'\rangle$:
        if $b' = b$ then
          $waitfor := waitfor - \{\alpha\}$;
          if $|waitfor| < |acceptors|/2$ then
            $\forall \rho \in replicas$:
              $send(\rho, \langle \mathbf{decision}, s, p\rangle)$;
            $exit()$;
          end if;
        else
          $send(\lambda, \langle \mathbf{preempted}, b'\rangle)$;
          $exit()$;
        end if;
      end case
    end switch
  end for
end process

Commander in multi-Paxos, in DistAlgo

```
class Commander extends Process:

   def setup(leader, acceptors, replicas, b, s, p): skip

   def run():

      send ('p2a', b, s, p) to acceptors

      await count({a: received(('p2b', =b) from a)}) > count(acceptors)/2:

         send ('decision', s, p) to replicas

      or received('p2b', b2) and b2!=b:

         send ('preempted', b2) to leader
```

no local update — synchronization condition is completely clear.

similar for Scout process in multi-Paxos

2. Direct high-level assertions

determining state is key to taking actions:

can assert state in many ways; need to test truth value of assertions as messages are sent and received

principles:

1. express assertions using queries over `received` and `sent`, as for synchronization conditions
2. use quantifications directly, vs loops and low-level updates
3. use quantifications directly, vs comprehensions and aggregates

example: conditions in Lamport's distributed mutex:

- request by self is before each other request in q
- an ack msg from each other proc is received after own request

Example: Lamport's distributed mutex

using quantifications directly:

```
each ('request',c2,p2) in q | (c2,p2)!=(c,self) implies (c,self) < (c2,p2)
and each p2 in s | some received('ack', c2, =p2) | c2 > c
```

using loops or updates: much more work, tedious and error-prone

using aggregates: `(c,self) < min({(c2,p2) in q})`
often incorrect and needs boundary values such as `maxint`,
even inefficient since min needs O(log n) update time,
but efficient incremental computation needs only O(1) time.

3. Straightforward high-level computations

computations are needed to achieve goals:
computations depend on messages sent and received;
need to compute results as messages are sent and received

principles:

1. compute aggregate values using aggregates over `received/sent`
2. compute set values using comprehensions over `received/sent`
3. specify repeated comps straightforwardly where results are used

example: acceptor in multi-Paxos:

- respond to `p1a` msgs from scouts with `p1b` msgs in phase 1
- respond to `p2a` msgs from commanders with `p2b` msgs in phase 2

Example: Acceptor in multi-Paxos [vR11]

```
process Acceptor()
  var ballot_num := ⊥, accepted := ∅;

  for ever
    switch receive()
      case ⟨p1a, λ, b⟩ :
        if b > ballot_num then
          ballot_num := b;
        end if;
        send(λ, ⟨p1b, self(), ballot_num, accepted⟩);
      end case
      case ⟨p2a, λ, ⟨b, s, p⟩⟩ :
        if b ≥ ballot_num then
          ballot_num := b;
          accepted := accepted ∪ {⟨b, s, p⟩};
        end if
        send(λ, ⟨p2b, self(), ballot_num⟩);
      end case
    end switch
  end for
end process
```

Acceptor in multi-Paxos, in DistAlgo

```
class Acceptor extends Process:
  def setup(): self.accepted = {}
  def run(): await false
  receive m:
    self.ballot_num = max({b: received('p1a',b)}+{b: received('p2a',b,_,_)} or {(-1,-1)})
  receive ('p1a', _) from scout:
    send ('p1b', ballot_num, accepted) to scout
  receive ('p2a', b, s, p) from commander:
    if b == ballot_num: accepted.add((b,s,p))
    send ('p2b', ballot_num) to commander
```

invariant for `ballot_num` is completely clear.

4. Collective send-actions

sending collections of msgs is generally needed to achieve goals:

algorithms should be viewed as driven by send-actions,
as opposed to by handling of individual received messages

method:

1. identify the kinds of messages to be sent
2. for each kind, collect all situations where the msgs are sent
3. express situations collectively using loops, favoring for-loops

example: replica in multi-Paxos:

- for each request received, send proposal to leaders until accepted
- for each acceptance, apply it to state and send result to client

Example: Replica in multi-Paxos [vR11]

```
process Replica(leaders, initial_state)
  var state := initial_state, slot_num := 1;
  var proposals := ∅, decisions := ∅;

  function propose(p)
    if ∄s : ⟨s, p⟩ ∈ decisions then
      s′ := min{s | s ∈ ℕ⁺ ∧
            ∄p′ : ⟨s, p′⟩ ∈ proposals ∪ decisions};
      proposals := proposals ∪ {⟨s′, p⟩};
      ∀λ ∈ leaders : send(λ, ⟨propose, s′, p⟩);
    end if
  end function

  function perform(⟨κ, cid, op⟩)
    if ∃s : s < slot_num ∧
            ⟨s, ⟨κ, cid, op⟩⟩ ∈ decisions then
      slot_num := slot_num + 1;
    else
      ⟨next, result⟩ := op(state);
      atomic
        state := next;
        slot_num := slot_num + 1;
      end atomic
      send(κ, ⟨response, cid, result⟩);
    end if
  end function

  for ever
    switch receive()
      case ⟨request, p⟩ :
        propose(p);
      case ⟨decision, s, p⟩ :
        decisions := decisions ∪ {⟨s, p⟩};
        while ∃p′ : ⟨slot_num, p′⟩ ∈ decisions do
          if ∃p″ : ⟨slot_num, p″⟩ ∈ proposals ∧
                  p″ ≠ p′ then
            propose(p″);
          end if
          perform(p′);
        end while;
    end switch
  end for
end process
```

Replica in multi-Paxos, in DistAlgo

```
class Replica extends Process:
  def setup(leaders, initial_state):
    self.state = initial_state
    self.slot_num = 1

  def run():
    while true:

      -- propose
      for ('request',p) in received:
        if each ('propose',s,=p) in sent | some received('decision',=s,p2) | p2!=p:
          s = min({s in 1.. max({s: sent('propose',s,_)}+{s: received('decision',s,_)})+1
                   | not (sent('propose',s,_) or received('decision',s,_))})
          send ('propose', s, p) to leaders

      -- perform
      while some ('decision', =slot_num, p) in received:
        if not some ('decision', s, =p) in received | s < slot_num:
          client, cmd_id, op = p
          state, result = op(state)
          send ('respond', cmd_id, result) to client
        slot_num += 1
```

conditions for send-actions are completely clear.

invariant for `slot_num` is completely clear.

Experiments with important algorithms

algorithms with interesting results and their sizes in DistAlgo:

Algorithm	Description	Spec size	Incr size
La mutex	Lamport's distributed mutual exclusion	32	43
2P commit	Two-phase commit	44	67
La Paxos	Lamport's Paxos for distributed consensus	43	59
CL Paxos	Castro-Liskov's Byzantine Paxos	63	81
vR Paxos	van Renesse's pseudocode for multi-Paxos	86	160

sizes are in number of lines excluding comments and empty lines.

Incr indicates specs containing low-level incremental updates;
for multi-Paxos, Incr size is for following pseudocode in [vR11].

compare with other languages:

La Paxos: 43 DistAlgo, 83 PlusCal, 145 IOA, 230 Overlog, 157 Bloom

vR Paxos: 86 DistAlgo, 130 pseudocode, ~3000 a Python implementation

Results for correctness & efficiency

La mutex:

algorithm simplified to not enqueue/dequeue own requests.

data structure for maintaining min request in O(log n) removed

2P commit:

succinct spec of coordinator: 2 awaits, 1 assertion, 1 set query

easy to see it is safe to add timeout to 1st wait, not 2nd wait

La Paxos and CR Paxos:

direct use of quantifications match English description.

our earlier uses of aggregates were incorrect or needed maxint.

vR Paxos:

for commander and scout, if / returns int, orig algo is incorrect.

for replica, re-proposals are delayed unnecessarily.

Generated implementations

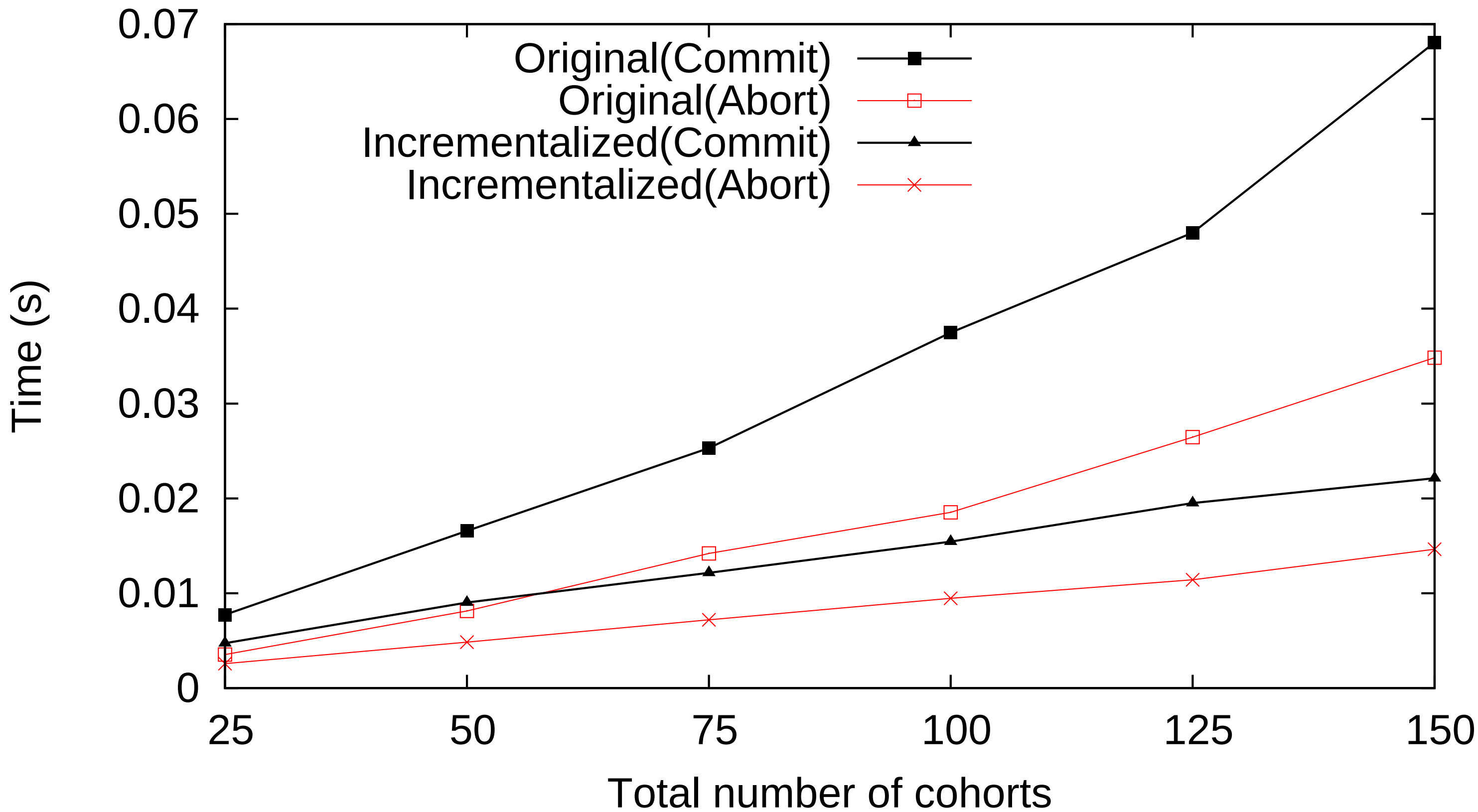
size of Python implementations generated from DistAlgo specs:

Algorithm	Spec size	Generated size
La mutex	32	1395
La mutex incr	43	1424
2P commit	44	1432
2P commit incr	67	1437
La Paxos	43	1428
La Paxos incr	59	1498
CL Paxos	63	1480
CL Paxos incr	81	1530
vR Paxos	86	1555
vR Paxos incr	160	1606

"incr" indicates specs containing low-level incremental updates.

compilation times are between 13 and 44 seconds.

Performance of generated implementation



for two-phase commit, for failure rates of 0 (Commit) and 100 (Abort), averaged over 50 rounds and 15 independent runs.

Grad and undergrad projects in DistAlgo

Project	Description	Notes
Leader	ring, randomized; arbitrary net	3 algorithms
Narada	overlay multicast system	
Chord	distributed hash table (DHT)	
Kademlia	DHT	
Pastry	DHT	
Tapestry	DHT	
HDFS	Hadoop distributed file system	part
UpRight	cluster services	part
AODV	wireless mesh network routing	python
OLSR	optimized link state routing	python

part: omitted replication, but done in our impl. of vR Paxos
python: in Python, but knew it would be easier in DistAlgo

each is about 300-600 lines, took about half a semester.

Summary and conclusion

use a simple and powerful language, DistAlgo: very high-level

- distributed processes as objects, sending messages
- yield points for control flow, handling of received messages

\+ await and synchronization conditions as queries of msg history

- high-level constructs for system configuration

exploit high-level abstractions of computation and control

1. high-level synchronization with explicit wait on received msgs
2. high-level assertions for when to send msgs and take actions
3. high-level queries for what to send in msgs to whom
4. collective send-actions for overall computation and control

experiment with important distributed algorithms

- including Paxos and multi-Paxos for distributed consensus
- discovered improvements to some, for correctness & efficiency

Future work

formal verification of higher-level algorithm specifications
by translating to PlusCal and other languages of verifiers

generating implementations in lower-level languages
C, Java, Erlang, ...

many additional, improved analyses and optimizations:
type analysis, deadcode analysis, cost analysis, ...

deriving optimized distributed algorithms
reducing message complexity and round complexity

Thanks!

Example: distributed mutual exclusion

Lamport's algorithm: developed to show logical timestamps

n processes access a shared resource, need mutex, go in CS

a process that wants to enter critical section (CS)

- send requests to all
- wait for replies from all
- enter CS
- send releases to all

each process maintains a queue of requests

- order by logical timestamps
- enter CS only if its request is the first on the queue
- when receiving a request, enqueue
- when receiving a release, dequeue

safety, liveness, fairness, efficiency

How to express it

two extremes, and many in between

1. English: clear high-level flow; imprecise, informal

2. state machine based specs: precise; low-level control flow
   Nancy Lynch's I/O automata: 1 1/5 pages, most two-column

in between:

- Michel Raynal's pseudocode: still informal and imprecise
- Leslie Lamport's PlusCal: still complex
  (90 lines excluding comments and empty lines, by Merz)
- Robbert van Renesse's pseudocode: precise, almost high-level

lack concepts for building real systems — much more complex
most of these are not executable at all.

Original description in English

The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.

1. To request the resource, process P_i sends the message $T_m : P_i$ *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.

2. When process P_j receives the message $T_m : P_i$ *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.

3. To release the resource, process P_i removes any $T_m : P_i$ *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.

4. When process P_j receives a P_i *releases resource* message, it removes any $T_m : P_i$ *requests resource* message from its request queue.

5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a $T_m : P_i$ *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $<$. (To define the relation $<$ for messages, we identify a message with the event of sending it.) (ii) P_i has received an acknowledgment message from every other process timestamped later than T_m.

Note that conditions (i) and (ii) of rule 5 are tested locally by P_i.

Challenges

each process must

- act as both P_i and P_j in interactions with all other processes
- have an order of handling all events by the 5 rules, trying to enter and exit CS while also responding to msgs from others
- keep testing the complex condition in rule 5 as events happen

actual implementations need many more details

- create processes, let them establish channels with each other
- incorporate appropriate clocks (e.g., Lamport, vector) if needed
- guarantee the specified channel properties (e.g., reliable, FIFO)
- integrate the algorithm with the overall application

how to do all of these in an easy and modular fashion?

- for both correctness verification and performance optimization

Original algorithm in DistAlgo

```
def setup(s):
  self.s = s                            # set of all other processes
  self.q = {}                           # set of pending requests with logical clock

def cs(task):                           # for calling task() in critical section
  -- request
  self.c = Lamport_clock()                                        # rule 1
  send ('request', c, self) to s                                  #
  q.add(('request', c, self))                                     #
  await each ('request',c2,p2) in q | (c2,p2) != (c,self) implies (c,self) < (c2,p2)
        and each p2 in s | some received('ack',c2,=p2) | c2 > c   # rule 5
  task()                                # critical section
  -- release
  q.del(('request', c, self))                                     # rule 3
  send ('release', Lamport_clock(), self) to s                    #

receive ('request', c2, p2):                                      # rule 2
  q.add(('request', c2, p2))                                      #
  send ('ack', Lamport_clock(), self) to p2                       #

receive ('release', _, p2):                                       # rule 4
  q.del(('request', _, =p2))                                      #
```

Complete program in DistAlgo

```
class P extends Process:

... # content of the previous slide

  def run():
    ...
    def task(): ...
    cs(task)
    ...

def main():
  ...
  use reliable_channel
  use fifo_channel
  use Lamport_clock
  ps = newprocesses(50,P)
  for p in ps: p.setup(ps-{p})
  for p in ps: p.start()
  ...
```

Optimized program after incrementalization

```
class P extends Process:
  def setup(s):
    self.s = s                                        # self.q was removed
    self.total = size(s)        # total number of other processes
    self.ds = new DS()          # aux DS for maint min of requests by other processes

  def cs(task):
    -- request
    self.c = Lamport_clock()
    self.responded = {}         # set of responded processes
    self.count = 0              # count of responded processes
    send ('request', c, self) to s                    # q.add(...) was removed
    await (ds.is_empty() or (c,self) < ds.min()) and count == total  # use maintained
    task()
    -- release
    send ('release', Lamport_clock(), self) to s  # q.del(...) was removed

  receive ('request', c2, p2):
    ds.add((c2,p2))             # add to the auxiliary data structure
    send ('ack', Lamport_clock(), self) to p2     # q.add(...) was removed

  receive ('ack', c2, p2):      # new message handler
    if c2 > c:                  # test comparison in condition 2
      if p2 in s:               # test membership in condition 2
        if p2 not in responded: # test whether responded already
          responded.add(p2)     # add to responded
          count += 1            # increment count

  receive ('release', _, p2):                         # q.del(...) was removed
    ds.del((_,=p2))             # remove from the auxiliary data structure
```

Simplified program by un-incrementalization

```
class P extends Process:
  def setup(s):
    self.s = s

  def cs(task):
    -- request
    self.c = Lamport_clock()
    send ('request', c, self) to s
    await each received('request',c2,p2) |
            not some received('release',c3,=p2) | c3 > c2 implies (c,self) < (c2,p2)
          and each p2 in s | some received('ack',c2,=p2) | c2 > c
    task()
    -- release
    send ('release', Lamport_clock(), self) to s

  receive ('request', _, p2):
    send ('ack', Lamport_clock(), self) to p2
```

Optimized w/o queue after incrementalization

```
class P extends Process:
  def setup(s):
    self.s = s
    self.q = {}                                 # self.q is kept as a set, no aux ds
    self.total = size(s)                        # total num of other processes

  def cs(task):
    -- request
    self.c = Lamport_clock()
    self.earlier = q                            # set of pending earlier reqs
    self.count1 = size(earlier)                 # num of pending earlier reqs
    self.responded = {}                         # set of responded processes
    self.count = 0                              # num of responded processes
    send ('request', c, self) to s
    q.add(('request', c, self))                 # q.add is kept, no aux ds.add
    await count1 == 0 and count == total        # use maintained results
    task()
    -- release
    q.del(('request', c, self))                 # q.del is kept,no aux ds.add
    send ('release', Lamport_clock(), self) to s

  receive ('request', c2, p2):
    if c != undefined:                          # if c is defined
      if (c,self) > (c2,p2):                    # test comparison in conjunct 1
        if ('request',c2,p2) not in earlier:    # if not in earlier
          earlier.add(('request',c2,p2))        # add to earlier
          count1 +=1                            # increment count1
    q.add(('request',c2,p2))                    # q.add is kept, no aux ds.add
    send ('ack', Lamport_clock(), self) to p2
```

```
receive ('ack', c2, p2):                        # new message handler
  if c2 > c:                                    # test comparison in conjunct 2
    if p2 in s:                                 # test membership in conjunct 2
      if p2 not in responded:                   # test whether responded already
        responded.add(p2)                       # add to responded
        count += 1                              # increment count

receive ('release', _, p2):
  if c != undefined:                            # if c is defined
    if (c,self) > (c2,p2):                      # test comparison in conjunct 1
      if ('request',c2,p2) in earlier:          # if in earlier
        earlier.del(('request',c2,p2))          # delete from earlier
      count1 -=1                                # decrement count1
  q.del(('request',_,=p2))                      # q.del is kept, no aux ds.del
```

Implementations of Lamport's algorithm

Language	Dist. programming features used	Total	Clean
C	TCP socket library	358	272
Java	TCP socket library	281	216
Python	multiprocessing package	165	122
Erlang	built-in message passing	177	99
PlusCal	single process simulation with array	134	90
DistAlgo	built-in high-level synchronization	48	32

program size in total number of lines of code,
and number of lines excluding comments and empty lines

Program size for well-known algorithms

Algorithm	DistAlgo	PlusCal	IOA	Overlog	Bloom
La mutex	32	90	64		
La mutex2	33				
RA mutex	35				
RA token	43				
SK token	42				
CR leader	30		41		
HS leader	56				
2P commit	44	68			85
DS crash	22				
La Paxos	43	83	145	230	157
CL Paxos	63	166			
vR Paxos	160				

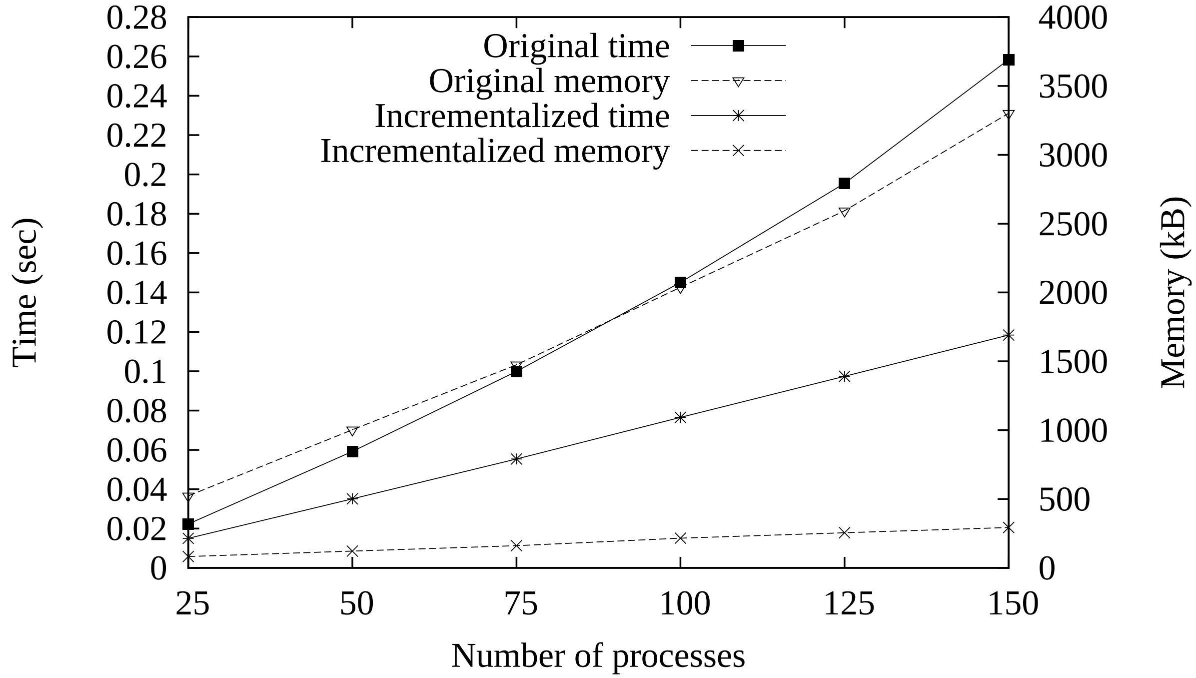
number of lines excluding comments and empty lines,
compared with specifications written by others in other languages

Compilation time and generated prog. sizes

Algorithm	Compilation time (ms)	DistAlgo size	Compiled size	Incremental-ized size
La mutex	13.3	32	1395	1424
La mutex2	15.3	33	1402	1433
RA mutex	12.3	35	1395	1395
RA token	12.9	43	1402	1402
SK token	16.5	42	1405	1407
CR leader	10.7	30	1395	1395
HS leader	18.7	56	1415	1415
2P commit	21.4	44	1432	1437
DS crash	10.5	22	1399	1414
La Paxos	20.7	43	1428	1498
CL Paxos	32.3	63	1480	1530
vR Paxos	43.4	160	1555	1606

compilation time not including incrementalization time (all $<$ 30s), and numbers of lines excluding comments and empty lines of generated programs (including 1300 lines of fixed library code)

Performance of generated implementation



running time and memory usage for Lamport's algorithm:

CPU time for each process to complete a call to `cs(task)`, including time spent handling messages from other processes, averaged over processes and over runs of 30 calls each;

raw size of all data structures created, measured using Pympler

Example: two-phase commit

a coordinator and a set of cohorts try to commit a transaction

phase 1:

- coordinator sends a prepare to all cohorts.
- each cohort replies with a ready vote if it is prepared to commit, or else replies with an abort vote and aborts.

phase 2:

- if coordinator receives a ready vote from all cohorts,
  it sends a commit to all cohorts;
  each cohort commits and sends a done to coordinator;
  coordinator completes when receives a done from all cohorts.

- if coordinator receives an abort vote from any cohort,
  it sends an abort to all cohorts who sent a ready vote;
  each cohort who sent a ready vote aborts.

agreement, validity, weak termination, 4n-4 msgs

How to express it

two extremes, and many in between

1. English: clear high-level flow; imprecise, informal

2. state machine based specs: precise; low-level control flow
   Nancy Lynch's I/O automata: book p183-184, but 2n-2 msgs

in between:

- Michel Raynal's pseudocode: still informal and imprecise

- Leslie Lamport's PlusCal: still complex
  (P2TwoPhase, 68 lines excluding comments and empty lines)

- Robbert van Renesse's pseudocode: precise, almost high-level

lack concepts for building real systems — much more complex
   most of these are not executable at all.

Original description in English

Phase 1: Summary of the protocol [KBL06 DB and TP]

1. The coordinator sends a *prepare message* to all cohorts.

2. Each cohort waits until it receives a *prepare message* from the coordinator. If it is prepared to commit, it forces a prepared record to its log, enters a state in which it cannot be aborted by its local control, and sends "ready" in the *vote message* to the coordinator.

If it cannot commit, it appends an abort record to its log. Or it might already have aborted. In either case, it sends "aborting" in the *vote message* to the coordinator, rolls back any changes the subtransaction has made to the database, release the subtransaction's locks, and terminates its participation in the protocol.

Phase 2:

1. The coordinator waits until it receives votes from all cohorts. If it receives at least one "aborting" vote, it decides to abort, sends an *abort message* to all cohorts that voted "ready", deallocates the transaction record in volatile memory, and terminates its participation in the protocol.

If all votes are "ready", the coordinator decides to commit (and stores that fact in the transaction record), forces a commit record (which includes a copy of the transaction record) to its log, and sends a *commit message* to each cohort.

2. Each cohort that voted "ready" waits to receive a message from the coordinator. If a cohort receives an *abort message*, it rolls back any changes the subtransaction has made to the database, appends an abort record to its log, releases the subtransaction's locks, and terminates it participation in the protocol.

If the cohort received a *commit message*, it forces a commit record to its log, releases all locks, sends a *done message* to the coordinator, and terminates its participation in the protocol.

3. If the coordinator committed the transaction, it waits until it receives *done message* from all cohorts. Then it appends a completion record to its log, deletes the transaction record from volatile memory, and terminates it participation in the protocol.

Original algorithm in DistAlgo

```
class Coordinator extends Process:
  def setup(tid, cohorts): pass  # transaction id and cohorts
  def run():
    send ('prepare',tid) to cohorts
    await each c in cohorts | received('vote',_,tid) from c
    if each c in cohorts | received('vote','ready',tid) from c:
      send ('commit',tid) to cohorts
      await each c in cohorts | received('done',tid) from c
      print(complete'+tid)
    else:
      s = {c in cohorts | received('vote','ready',tid) from c}
      send ('abort',tid) to s
    print('terminate'+tid)

class Cohort extends Process:
  def setup(f): pass  # failure rate
  def run():
    await(False)
  receive ('prepare',tid) from c:
    if prepared(tid):
      send ('vote','ready,tid) to c     # await commit or abort here?
    else:
      send ('vote','abort',tid) to c
      abort(tid)
  receive ('commit',tid) from c:
    commit(tid)
    send ('done',tid) to c
  receive ('abort',tid):
    abort(tid)

def prepared(tid): return randint(0,100) > f
def abort(tid): print('abort'+tid)
def commit(tid): print('commit'+tid)
```

Complete program in DistAlgo

```
from random import randint

... # content of the previous slide

def main():
  cs = createprocs(Cohort,25,(10))       # create 25 cohorts
  c = createprocs(Coordinator,1,(0,cs))  # create 1 coordinator
  startprocs(cs)                         # start cohorts
  startprocs(c)                          # start coordinator
```

Optimized after incrementalization (part 1)

```
class Coordinator extends Process:

  def setup(tid, cohorts):
    ncohorts = size(cohorts)  # number of cohorts
    svoted = {}               # set of voted cohorts
    nvoted = 0                # number of voted cohorts
    sready = {}               # set of ready cohorts
    nready = 0                # number of ready cohorts
    sdone = {}                # set of done cohorts
    ndone = 0                 # number of done cohorts

  def run():
    send ('prepare',tid) to cohorts
    await nvoted == ncohorts  # replaced universal quantification
    if nready == ncohorts:    # replaced universal quantification
      send ('commit',tid) to cohorts
      await ndone == ncohorts # replaced universal quantification
      print('complete'+tid)
    else:
      s = sready              # replaced set query
      send ('abort',tid) to s
    print('terminate'+tid)
```

Optimized after incrementalization (part 2)

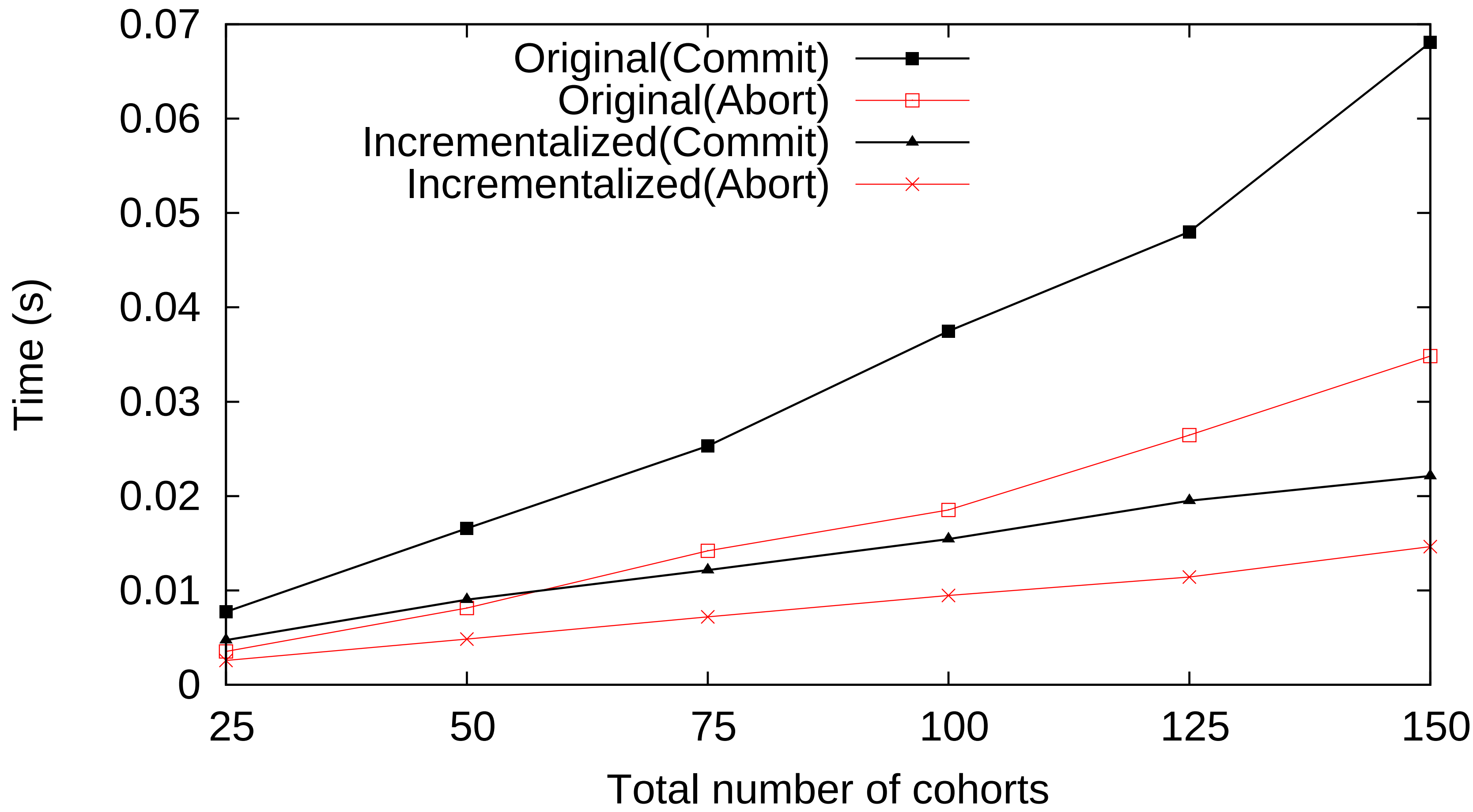
```
    # new message handler
    receive ('vote',v,tid) from c:
      if c in cohorts:
        if c not in svoted:
          svoted.add(c)
          nvoted += 1
        if v == 'ready':
          if c not in sready:
            sready.add(c)
            nready += 1

    # new message handler
    receive ('done',tid) from c:
      if c in cohorts:
        if c not in sdone:
          sdone.add(c)
          ndone += 1

class Cohort extends Process:
  ... # no change

def main():
  ... # no change
```

Performance of generated implementation



for two-phase commit, for failure rates of 0 (Commit) and 100 (Abort), averaged over 50 rounds and 15 independent runs.

Expensive queries using quantifications

expensive computation of synchronization condition:

```
each ('request',c2,p2) in q | (c2,p2) != (c,self) implies (c,self) < (c2,p2)
and each p2 in s | some received('ack',c2,p2) | c2 > c
```

all updates to variables used by expensive computations:

```
 2     self.s = s
 3     self.q = {}

 7     self.c = Lamport_clock()
 8     q.add(('request', c, self))
13     q.del(('request', c, self))
16     q.add(('request', c2, p2))
19     q.del(('request', _, p2))

 *     received.add(('ack',c2,p2))
```

transform queries into efficient incremental computation at updates

how?

Optimization by incrementalization

- introduce variables to store values of queries
- transform the queries to use introduced variables
- incrementally maintain stored values at each update

new: systematic handling of

1. quantifications for synchronization as expensive queries

2. updates caused by sending, receiving, and handling of msgs in the same way as other updates in the program

transform expensive synchronization conditions into efficient tests and incremental updates as msgs are sent and received

sequences `received` and `sent` will be removed as appropriate
only values needed for incremental computation of synchronization conditions will be stored and incrementally updated

Incrementalization of quantifications

transform quantifications into aggregates:

```
({(c2,p2) : ('request',c2,p2) in q | (c2,p2) != (c,self)} == {} or
 (c,self) < min({(c2,p2) : ('request',c2,p2) in q | (c2,p2) != (c,self)}))
and
size({p2: p2 in s, ('ack',c2,=p2) in received | c2 > c})  ==  size(s)
```

without queue:

```
size({('request',c2,p2) in q | (c,self) > (c2,p2)})  ==  0 and ...
```

use incrementally maintained query results:

```
(ds.is_empty() or (c,self) < ds.min()) and count == total
```

without queue:

```
count1 == 0 and ...
```

use max and min if no deletion — maintain single value, not set