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(λ, acceptors, replicas, ⟨b, s, p⟩)
  var waitfor := acceptors;

  ∀α ∈ acceptors : send(α, ⟨p2a, self(), ⟨b, s, p⟩⟩);
  for ever
    switch receive()
      case ⟨p2b, α, b′⟩ :
        if b′ = b then
          waitfor := waitfor − {α};
          if |waitfor| < |acceptors|/2 then
            ∀ρ ∈ replicas :
              send(ρ, ⟨decision, s, p⟩);
            exit();
          end if;
        else
          send(λ, ⟨preempted, b′⟩);
          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

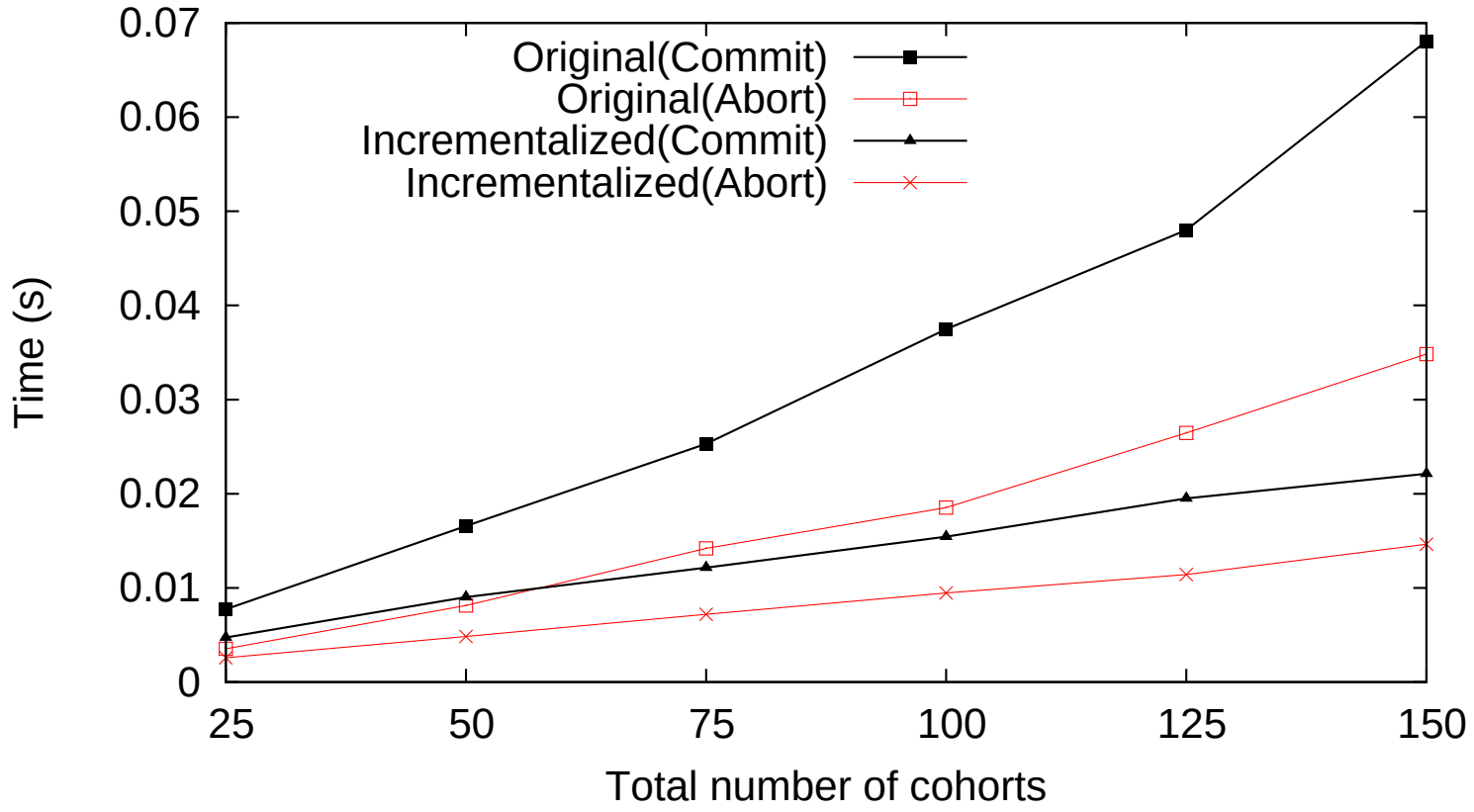
size of Python implementations generated from DistAlgo specs:

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