Invariants in Distributed Algorithms

Y. Annie Liu, Scott D. Stoller

Computer Science Department
Stony Brook University

joint work with
Saksham Chand, Bo Lin, and Xuetian Weng

Distributed algorithms and correctness

distributed systems: increasingly important and complex

everyday life: search engines, social networks, electronic commerce, cloud computing, mobile computing, ...

distributed algorithms: increasingly needed and complex

for distributed control and distributed data, e.g., distributed consensus, DHT, ...

correctness guarantees: increasingly needed and challenging

safety, liveness, fairness, ..., improved guarantees

Expressing and understanding algorithms

need languages

- pseudocode languages and English — high-level

  in many textbooks and papers

- specification languages — precise

  TLA and PlusCal by Lamport,
  IOA and TIOA by Lynch's group, ...

- programming languages — executable

  Argus by Liskov's group, Emerald, Erlang, ...
  libraries in C, C++, Java, Python, ...: socket, MPI, ...

DistAlgo: combines advantages of all three [TOPLAS 2017]

Overview

DistAlgo: expressing, understanding, optimizing, and improving distributed algorithms

example: Lamport's algorithm for distributed mutual exclusion

verification: formal semantics, translation to TLA+

proofs using TLAPS: Paxos for distributed consensus
model checking using TLC: Lamport's distributed mutex

invariants: clear specs, optimization, improvement, easier proofs

through high-level queries over history variables

Lamport's distributed mutual exclusion

Lamport developed it to show the logical clocks he invented

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

requests must be granted in the order in which they are made

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

reliable, fifo channel — safety, liveness, fairness, efficiency

requests are granted in the order of timestamps of requests

How to express it

two extremes:

- English: clear high-level flow; imprecise, informal
- state machine based specs: precise; low-level control flow
  e.g., Nancy Lynch's I/O automata (1 1/5 pages, most 2-col.)

many in between, e.g.:

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

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

Lamport's 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.

order $<$ on requests: pairs of logical time and process id.

There will be an interesting exercise later, if there is time.

Challenges in expressing it

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

DistAlgo language

as extensions to common high-level languages
    including a syntax for extensions to Python

distributed processes and sending messages

- `process` P`: ...` — define `setup(`$pars$`)`, `run()`, `receive`
- `send` ms `to` ps

control flows and receiving messages

- `--` l`:` — yield point for handling msgs
- `receive` m `from` p`: ...` — handler
- `await` $cond_1$`: ... or...or` $cond_k$`: ... timeout` t`: ...`

high-level queries of message histories

- `some` v_1 `in` s_1`,...,`v_k `in` s_k `has` $cond$ — also `each`/set/min
- `received` m is same as m `in received`

configurations

- `configure clock = Lamport`
- ps `:=` n `new` P — call `setup`/`start`

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 mutex(task):                          # for doing task() in critical section
  -- request
  self.t := logical_time()                                  # rule 1
  send ('request', t, self) to s                            #
  q.add(('request', t, self))                               #
  await each ('request',t2,p2) in q | (t2,p2) != (t,self) implies (t,self) < (t2,p2)
        and each p2 in s | some received ('ack',t2,=p2) | t2 > t  # rule 5
  task()                                  # critical section
  -- release
  q.del(('request', t, self))                               # rule 3
  send ('release', logical_time(), self) to s               #

receive ('request', t2, p2):                                # rule 2
  q.add(('request', t2, p2))                                #
  send ('ack', logical_time(), self) to p2                  #

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

Complete program in DistAlgo

```
process P:

    ... # content of the previous slide

    def run():
      def task(): output(self, 'in critical section')
      mutex(task)

def main():
  configure clock = Lamport
  configure channel = {reliable, fifo}
  ps := 50 new P
  for p in ps: p.setup(ps-{p})
  ps.start()
```

some syntax in Python:

```
class P( process )
send( m, to= ps )
some( elem in s, has= bexp )
config( clock= 'Lamport' )
new( P, num= 50 )
```

Formal operational semantics

Reduction semantics with evaluation contexts
for a core language for DistAlgo

- Traditional constructs
  - Booleans, integers, addresses
  - class definition, object creation, method call, ...
  - `if`, `while`, `for` (over sets), assignment, ...
- DistAlgo constructs
  - `start`, `send`, `receive` handlers, `await`
  - set comprehension and quantifications with tuple patterns in membership clauses

Some constructs (e.g., tuple patterns, set comprehensions) are given semantics by translation.

Formal semantics: Overview

state: local state of each process + message channel contents

local state: heap + statement remaining to be executed

evaluation context: identifies the sub-expression or sub-statement to be evaluated next

transition: updates the statement (e.g., removes the part just executed, unrolls a loop, or inlines a method call), the local heap, and the message channel contents

execution: sequence of transitions starting from an initial state

- may terminate, get stuck, or continue forever

Formal semantics: Evaluation context

evaluation context: an expression or statement with a hole, denoted `[]`, in place of the next sub-expression or sub-statement to be evaluated.

C ::= []

$(Val*, C, Expression*)$

$C.MethodName(Expression*)$

$Address.MethodName(Val*, C, Expression*)$

$UnaryOp(C)$

some *Pattern* **in** C **|** *Expression*

if C**:** *Statement* **else:** *Statement*

for *InstanceVariable* **in** C**:** *Statement*

send C **to** *Expression*

send *Val* **to** C

await *Expression* **:** *Statement AnotherAwaitClause** **timeout** C

...

Formal semantics: Transition relation

$\sigma \to \sigma'$ state σ can transition to state σ'.

state: a tuple of the form (P, ht, h, ch, mq)

P: map from process address to remaining statement

h: heap, ht: heap type map

ch: message channel contents (messages in transit)

mq: message queue contents (arrived, unhandled messages)

sample transition rule

// context rule for statements

$$\frac{(P[a \to s], ht, h, ch, mq) \to (P[a := s'], ht', h', ch', mq')}{(P[a \to C[s]], ht, h, ch, mq) \to (P[a := C[s']], ht', h', ch', mq')}$$

Transition rule for handling messages

// handle a message at a yield point. remove the
// (message, sender) pair from the message queue, append a
// copy to the `received` sequence, and prepare to run
// matching receive handlers associated with ℓ, if any.
// s has a label hence must be `await`.

$$(P[a \rightarrow \ell\ s], ht, h[a \rightarrow ha], ch, mq[a \rightarrow q])$$

$$\begin{aligned}\rightarrow (&P[a := s'[\texttt{self} := a]; \ell\ s],\\ &ht', h[a \rightarrow ha'[a_r \rightarrow ha(a_r)@\langle copy\rangle]],\\ &ch, mq[a := rest(q)])\end{aligned}$$

$$\begin{aligned}\text{if } &length(q) > 0 \wedge a_r = ha(a)(\texttt{received})\\ &\wedge isCopy(first(q), ha, ha, ht, copy, ha', ht')\\ &\wedge receiveAtLabel(first(q), \ell, ht(a), ha') = S\\ &\wedge s' \text{ is a linearization of } S\end{aligned}$$

Transition rule for starting a process

// `process.start` allocates a local heap and `sent` and `received`
// sequences for the new process, and moves the started
// process to the new local heap.

$$
\begin{array}{l}
(P[a \to a'.\mathtt{start}()], ht, h[a \to ha[a' \to o], ch, mq) \\
\to (P[a := \mathtt{skip}, a' := a'.\mathtt{run}()], ht[a_s := \mathtt{sequence}, a_r := \mathtt{sequence}], \\
\quad h[a := ha \ominus a', a' := f_0[a' \to o[\mathtt{sent} := a_s, \mathtt{received} := a_r], \\
\qquad\qquad\qquad\qquad a_r := \langle\rangle, a_s := \langle\rangle]], \\
\quad ch, mq) \\
\text{ if } \mathit{extends}(ht(a'), \mathtt{process}) \wedge (ht(a') \text{ inherits } \mathtt{start} \text{ from } \mathtt{process}) \\
\quad \wedge\, a_r \notin \mathit{dom}(ht) \wedge a_s \notin \mathit{dom}(ht) \\
\quad \wedge\, a_r \in \mathit{NonProcessAddress} \wedge a_s \in \mathit{NonProcessAddress}
\end{array}
$$

Formal verification: Translation to TLA+

manual specification: for using TLC and TLAPS at all
  Basic Paxos, Multi, Fast, Vertical: checking using TLC
  Multi-Paxos, Multi-Paxos with Preemption, minimally ext.
    Lamport et al's Basic Paxos: safety proof in TLAPS

manual translation: for safety proof of more complex Paxos
  Multi with Preemption, state reduction, failure detection

automatic translation: from
  first: Python parser AST, second: own parser AST,
  last: Python parser own AST
  ongoing: DistAlgo actions—a DistAlgo subset

Model checking using TLC

using manual specification:

- checking small number of processes, simpler algorithms:
  Basic Paxos, Fast Paxos, Vertical Paxos: 3 acceptors...
- too slow for more complex algorithms or more processes:
  Multi-Paxos, $>$ 3 processes...
- did not find any violations even when there was
  a more complex variant of Multi-Paxos

using automatically translated: from much worse to worse

- first: each DistAlgo construct into 1 or more TLA+ actions
- last: use low-level intermediate rep. and compiler opts

Lamport's distributed mutex, number of states:

- Lamport TLA+: 28,358. our generated with last: 37,978
- Merz TLA+: 1,180,688. our generated with last: 2,052,276

Summary

DistAlgo: expressing, understanding, optimizing, and improving distributed algorithms

example: Lamport's algorithm for distributed mutual exclusion

verification: formal semantics, translation to TLA+

proofs using TLAPS: Paxos for distributed consensus
model checking using TLC: Lamport's distributed mutex

invariants: clear specs, optimization, improvement, easier proofs

through high-level queries over history variables

Invariants in distributed algorithms

high-level queries over history variables, allowing

clear specifications:
use high-level queries for synchronization conditions

optimization by incrementalization:
transform expensive queries into incremental updates

algorithm improvements:
simplified and improved algorithms (correctness and efficiency)

easier proofs:
need fewer manually written invariants

Lamport's dist. mutex: Simplified, improved

Original. in DistAlgo, at same high level as Lamport's English, except operations of both P_i and P_j are operations of P

Send-to-self. in 1&3, P_i need not enqueue/dequeue own request, but send request/release to all incl. self. 2&4 does enq/deq.

Inc-with-queue. expensive conditions (i)&(ii) in 5 are optimized by incremental maintenance as messages are received, incl. using dynamic queue for minimum of other reqs in (i).

Ignore-self. discovered in Inc-with-queue, in 1&3, P_i need not enqueue/dequeue own request or send request/release to self. (i) in 5 compares only with other requests anyway.

Inc-without-queue. (i) in 5 is better optimized by inc. maint., by using just a count of requests $<$ own request, and using a bit for each process if messages can be duplicated.

Simplified. discovered in Inc-with-queue and Inc-without-queue, (i) in 5 can just compare with request for which a release has not been received, omitting all updates of queue in 1-4.

Lamport's dist. mutex: Improved fairness

further simplifications:

remove unnecessary uses of logical clocks

improved understanding of fairness

use of any ordering for fairness:

including improved fairness
for granting requests in the order they are made,
over using logical clock values

discovery that logical clocks are not fair in general

exercise: for Lamport's mutex, if follow original English exactly,
easy to see safety and liveness violations too

Paxos made moderately complex: simplified and improved

Paxos made moderately complex [vRA 2015-ACMCS]:

Multi-Paxos with preemption, reconfiguration, state reduction, and failure detection

simplified specification: total about 50 lines

without scattered updates, from already greatly reduced

found errors and improvements:

previously unknown
useless replies, unnecessary delays, a liveness violation

and a safety violation in an earlier spec of ours

through TLAPS proof effort! after several years of teaching, with special efforts in testing and model checking

References

DistAlgo language and optimization [OOPSLA 2012/TOPLAS 2017]

implementation [OOPSLA 2012] formal semantics [TOPLAS 2017]

high-level executable specifications of distributed algorithms [SSS 2012]

TLA specification and TLAPS proofs of Multi-Paxos [FM 2016]

TLA specification and TLAPS proofs using history variables [NFM 2018]

moderated complex Paxos made simple [arXiv 2017/18]

logical clocks are not fair [APPLIED 2018]

DistAlgo resources

`http://github.com/DistAlgo` `http://distalgo.sourceforge.net`

`README`

can download — unzip — run script without installation
or to install: add to python path or run `python setup.py install`

or not even download if you have pip: run `pip install pyDistAlgo`

`http://distalgo.cs.stonybrook.edu`

tutorial (to update)
language description
formal operational semantics

more example algorithms given with DistAlgo implementation
among a wide variety of algorithms and protocols in DistAlgo, including core of many distributed systems and services in dozens of different course projects by hundreds of students

Ongoing and future work

easier and simpler specifications

DistAlgo actions: DistAlgo subset corresp. to TLA actions

more automated proofs

direct translation to TLA+
automated proof by induction: corresp. to incrementalization

many additional, improved analyses and optimizations:

type analysis, deadcode analysis, cost analysis, ...
efficient C/Erlang implementation, ... new algorithms

languages for more advanced computations:

security protocols, probabilistic inference, ...

Thanks !

Optimized w/ queue 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 mutex(task):
    -- request
    self.t := logical_time()
    self.responded := {}        # set of responded processes
    self.count := 0             # count of responded processes
    send ('request', t, self) to s                       # q.add(...) was removed
    await (ds.is_empty() or (t,self) < ds.min()) and count = total   # use maintained
    task()
    -- release
    send ('release', logical_time(), self) to s   # q.del(...) was removed

  receive ('request', t2, p2):
    ds.add((t2,p2))             # add to the auxiliary data structure
    send ('ack', logical_time(), self) to p2      # q.add(...) was removed

  receive ('ack', t2, p2):      # new message handler
    if t2 > t:                  # 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
```

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 mutex(task):
    -- request
    self.t = logical_time()
    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', t, self) to s
    q.add(('request', t, self))                # q.add is kept, no aux ds.add
    await count1 = 0 and count = total         # use maintained results
    task()
    -- release
    q.del(('request', t, self))                # q.del is kept,no aux ds.add
    send ('release', logical_time(), self) to s

  receive ('request', t2, p2):
    if t != undefined:                         # if t is defined
      if (t,self) > (t2,p2):                   # test comparison in conjunct 1
        if ('request',t2,p2) not in earlier:   # if not in earlier
          earlier.add(('request',t2,p2))       # add to earlier
          count1 +:= 1                         # increment count1
    q.add(('request',t2,p2))                   # q.add is kept, no aux ds.add
    send ('ack', logical_time(), self) to p2
```

```
receive ('ack', t2, p2):                          # new message handler
  if t2 > t:                                      # 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 t != undefined:                              # if t is defined
    if (t,self) > (t2,p2):                        # test comparison in conjunct 1
      if ('request',t2,p2) in earlier:            # if in earlier
        earlier.del(('request',t2,p2))            # delete from earlier
        count1 -:=1                               # decrement count1
  q.del(('request',_,=p2))                        # q.del is kept, no aux ds.del
```

Simplified algorithm

```
process P:
  def setup(s):
    self.s := s

  def mutex(task):
    -- request
    self.t = logical_time()
    send ('request', t, self) to s
    await each received ('request',t2,p2) |
            not (some received ('release',t3,=p2) | t3 > t2) implies (t,self) < (t2,p2)
          and each p2 in s | some received ('ack',t2,=p2) | t2 > t
    task()
    -- release
    send ('release', logical_time(), self) to s

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

eliminated all updates of queue

by un-incrementalization

Further simplified algorithm (1/2)

```
process P:
  def setup(s):
    self.s := s

  def mutex(task):
    -- request
    self.t := logical_time()
    send ('request', t, self) to s
    await each received ('request',t2,p2) |
            not received ('release',t2,p2) implies (t,self) < (t2,p2)
          and each p2 in s | some received ('ack',t2,=p2) | t2 > t
    task()
    -- release
    send ('release', t, self) to s

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

removed unnecessary use of logical times in release messages

Further simplified algorithm (2/2)

```
process P:
  def setup(s):
    self.s := s

  def mutex(task):
    -- request
    self.t := logical_time()
    send ('request', t, self) to s
    await each received ('request',t2,p2) |
            not received ('release',t2,p2) implies (t,self) < (t2,p2)
          and each p2 in s | received ('ack',t,p2)
    task()
    -- release
    send ('release', t, self) to s

  receive ('request', t2, p2):
    send ('ack', t2, self) to p2
```

removed unnecessary use of logical times in ack messages

logical times are used only in request messages

DistAlgo language overview

as extensions to common object-oriented languages

including a syntax for extensions to Python

1. distributed processes and sending messages

2. control flows and receiving messages

3. high-level queries of message histories

4. configurations

1. Distributed processes, sending messages

process definition

process p**:** *process_body* — **setup**, **run**, **self**

class p **(process):** *process_body*

process creation, setup, and start

v **=** n **new** p **at** *node_exp*

v **= new(**p**, at =** *node_exp***, num =** n**)**

*pexp***.setup(***args***)**

setup(*pexp***, (***args***))**

*pexp***.start()**

start(*pexp***)**

sending messages (usually tuples)

send *mexp* **to** *pexp*

send(*mexp***, to =** *pexp***)**

2. Control flows, receiving messages

yield point with label

```
-- l:
-- l
```

handling messages received

receive *mexp* from *pexp* at l_1,...,l_j:
    handler_body

def receive(msg = *mexp*, from_ = *pexp*, at = (l_1,...,l_j)):
    handler_body

synchronization (nondeterminism)

await *bexp*

await(*bexp*)

await $bexp_1$: $stmt_1$ or ... or $bexp_k$: $stmt_k$

timeout *t*: *stmt*

if await($bexp_1$): $stmt_1$ elif ... elif $bexp_k$: $stmt_k$

elif timeout(*t*): *stmt*

3. High-level queries of message histories

message sequences: `received`, `sent`

`received` *mexp* `from` *pexp*

mexp `from` *pexp* `in received`

`received(`*mexp*`, from_ = `*pexp*`)`

`(`*mexp*`, `*pexp*`) in received`

1) comprehensions

$\{$*exp*`:` v_1 `in` $sexp_1$`, ..., `v_k `in` $sexp_k$`, `*bexp*$\}$

`setof(`*exp*`, `v_1 `in` $sexp_1$`, ..., `v_k `in` $sexp_k$`, `*bexp*`)`

2) aggregates

agg_op comprehension_exp

agg_op`(`*comprehension_exp*`)`

3) quantifications

`some` v_1 `in` $sexp_1$`, ..., `v_k `in` $sexp_k$ `has` *bexp*

`each` v_1 `in` $sexp_1$`, ..., `v_k `in` $sexp_k$ `has` *bexp*

`some(`v_1 `in` $sexp_1$`, ..., `v_k `in` $sexp_k$`, has = `*bexp*`)`

`each(`v_1 `in` $sexp_1$`, ..., `v_k `in` $sexp_k$`, has = `*bexp*`)`

tuple patterns, left side of membership clause

4. Configurations

channel types

```
configure channel = fifo
config(channel = 'fifo')
```

default is not FIFO or reliable

message handling

```
configure handling = all
config(handling = 'all')
```

this is the default

logical clocks

```
configure clock = Lamport
config(clock = 'Lamport')
```

call `logical_time()` to get the logical time

overall: `.da` files

process definitions, method `main`, and conventional parts;

`main`: configurations and process creation, setup, and start