From Clarity to Efficiency for Distributed Algorithms

Y. Annie Liu

Computer Science Department
State University of New York at Stony Brook

joint work with
Scott Stoller, Bo Lin, and Michael Gorbovitski
Age of distributed programming

- search engines
- social networks
- cloud computing
- mobile computing
Programming algorithms

significant advances in programming languages:
  ... ALGOL ... C++ ... Java ... Python ... Prolog ...

• statements: assignments, conditionals, loops
• expressions: arithmetic, Boolean, other data (sets)
• subroutines: functions, procedures (recursion)
• logic rules: predicates, deduction, though less used
• objects: keep data and do operations (organization)

that's mostly sequential and centralized.
Concurrent programming

threads: multiple threads accessing shared data

threads as concurrent objects
  • the concurrent programming model in Java
  • adopted by other languages, such as Python and C#

Java made concurrent programming easier.
distributed programming

low-level or complex libraries, or restricted programming models

- sockets: C, Java, ... most widely used languages
- MPI: Fortran, C++, ... for high performance computing
- RPC: C, ... just about any language, Java RMI
- processes: Erlang, and more theoretically studied languages

... study of distributed algorithms, not for building real applications

- pseudocode, English: high-level but imprecise, not executable
- formal specification languages: precise but lower-level

much less progress
Our work: DistAlgo

a simple and powerful new language: very high-level, executable
- distributed processes as objects, sending messages
- yield points for control flow, handling messages
- await and synchronization conditions as queries of msg history
- high-level constructs for system configuration

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

experiments with well-known distributed algorithms
- including Paxos and multi-Paxos for distributed consensus
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.
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 $\prec$. (To define the relation $\prec$ 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
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
    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

    send ('release', Lamport_clock(), self) to s  # rule 3

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

0 class P extends Process:

... # content of the previous slide

20 def run():
   ...
21 def task(): ...
22 cs(task)
   ...

23 def main():
   ...
24 use reliable_channel
25 use fifo_channel
26 use Lamport_clock
27 ps = newprocesses(50,P)
28 for p in ps: p.setup(ps-{p})
29 for p in ps: p.start()  ...
class P extends Process:

    def setup(s):
        self.s = s
        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):
        self.c = Lamport_clock()
        self.responded = {}
        self.count = 0
        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()
        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:
            if p2 in s:
                if p2 not in responded:
                    responded.add(p2)
                    count += 1

    receive ('release', _, p2):  # q.del(...) was removed
        ds.del((_,=p2))  # remove from the auxiliary data structure
Simplified program by un-incrementalization

```python
0 class P extends Process:
1    def setup(s):
2        self.s = s

3    def cs(task):
4        -- request
5        self.c = Lamport_clock()
6        send ('request', c, self) to s
7        await each received('request',c2,p2) |
8            not some received('release',c3,=p2) | c3 > c2 implies (c,self) < (c2,p2)
9            and each p2 in s | some received('ack',c2,=p2) | c2 > c
10       task()
11    -- release
12    send ('release', Lamport_clock(), self) to s

12    receive ('request', _, p2):
13    send ('ack', Lamport_clock(), self) to p2
```
Optimized w/o queue after incrementalization

```python
0 class P extends Process:
1    def setup(s):
2        self.s = s
3        self.q = {}
4        self.total = size(s)
5
6    def cs(task):
7        -- request
8        self.c = Lamport_clock()
9        self.earlier = q
10       self.count1 = size(earlier)
11       self.responded = {}
12       self.count = 0
13       send ('request', c, self) to s
14       q.add(('request', c, self))
15       await count1 == 0 and count == total
16       task()
17        -- release
18       q.del(('request', c, self))
19       send ('release', Lamport_clock(), self) to s
20
21    receive ('request', c2, p2):
22        if c != undefined:
23            if (c, self) > (c2, p2):
24                if ('request', c2, p2) not in earlier:
25                    earlier.add(('request', c2, p2))
26                    count1 += 1
27                    q.add(('request', c2, p2))
28            send ('ack', Lamport_clock(), self) to p2
```
receive ('ack', c2, p2):
    if c2 > c:
        if p2 in s:
            if p2 not in responded:
                responded.add(p2)
                count += 1

receive ('release', _, p2):
    if c != undefined:
        if (c, self) > (c2, p2):
            if ('request', c2, p2) in earlier:
                earlier.del(('request', c2, p2))
                count1 -= 1
            q.del(('request', _, p2))
# Implementations of Lamport’s algorithm

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

Program size in total number of lines of code, and number of lines excluding comments and empty lines.
Program size for well-known algorithms

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

number of lines excluding comments and empty lines, compared with specifications written by others in other languages
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Compilation time (ms)</th>
<th>DistAlgo size</th>
<th>Compiled size</th>
<th>Incremental-ized size</th>
</tr>
</thead>
<tbody>
<tr>
<td>La mutex</td>
<td>13.3</td>
<td>32</td>
<td>1395</td>
<td>1424</td>
</tr>
<tr>
<td>La mutex2</td>
<td>15.3</td>
<td>33</td>
<td>1402</td>
<td>1433</td>
</tr>
<tr>
<td>RA mutex</td>
<td>12.3</td>
<td>35</td>
<td>1395</td>
<td>1395</td>
</tr>
<tr>
<td>RA token</td>
<td>12.9</td>
<td>43</td>
<td>1402</td>
<td>1402</td>
</tr>
<tr>
<td>SK token</td>
<td>16.5</td>
<td>42</td>
<td>1405</td>
<td>1407</td>
</tr>
<tr>
<td>CR leader</td>
<td>10.7</td>
<td>30</td>
<td>1395</td>
<td>1395</td>
</tr>
<tr>
<td>HS leader</td>
<td>18.7</td>
<td>56</td>
<td>1415</td>
<td>1415</td>
</tr>
<tr>
<td>2P commit</td>
<td>21.4</td>
<td>44</td>
<td>1432</td>
<td>1437</td>
</tr>
<tr>
<td>DS crash</td>
<td>10.5</td>
<td>22</td>
<td>1399</td>
<td>1414</td>
</tr>
<tr>
<td>La Paxos</td>
<td>20.7</td>
<td>43</td>
<td>1428</td>
<td>1498</td>
</tr>
<tr>
<td>CL Paxos</td>
<td>32.3</td>
<td>63</td>
<td>1480</td>
<td>1530</td>
</tr>
<tr>
<td>vR Paxos</td>
<td>43.4</td>
<td>160</td>
<td>1555</td>
<td>1606</td>
</tr>
</tbody>
</table>

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)
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
Grad and undergrad projects in DistAlgo

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

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 of code, took about half a semester.
Summary and conclusion

programming distributed algorithms: much to do
need both clarity (both high-level and precise) and efficiency

DistAlgo: a new language, simple, powerful
distributed processes and sending messages
yield points and handling messages
await and synchronization conditions as queries of msg history
high-level constructs for system configuration

powerful new optimization
transform expensive synchronization conditions
into efficient handlers as messages are sent and received,
by incrementalizing queries, especially logic quantifications
via incremental aggregate ops on sophisticated data structures

experiments with well-known distributed algorithms
including Paxos and multi-Paxos for distributed consensus
Thanks!
DistAlgo: distributed procs and sending msgs

process definition

    class P extends Process: class_body
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 and 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 l: stmt
allows handling of message m at label l; default is at all labels

synchronization

await bexp timeout time
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
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:

\[
\{(c_2,p_2) : ('request',c_2,p_2) \in q \mid (c_2,p_2) \neq (c,\text{self})\} = \{\} \text{ or } (c,\text{self}) < \min(\{(c_2,p_2) : ('request',c_2,p_2) \in q \mid (c_2,p_2) \neq (c,\text{self})\})
\]

and

\[
\text{size}(\{p_2: p_2 \in s, ('ack',c_2,\text{=}p_2) \in \text{received} \mid c_2 > c\}) = \text{size}(s)
\]

without queue:

\[
\text{size}(\{(‘request’,c_2,p_2) \in q \mid (c,\text{self}) > (c_2,p_2)\}) = 0 \text{ and } ...
\]

use incrementally maintained query results:

\[
(\text{ds.is_empty()} \text{ or } (c,\text{self}) < \text{ds.min()} \text{ and } \text{count} == \text{total})
\]

without queue:

\[
\text{count1} == 0 \text{ and } ...
\]

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

**syntax and parsing**: build on python parser.
provide two options for yield points and message handlers:
- modify python ASDL to take succinct syntax
- use unmodified python exploiting existing python syntax

**compilation**:
- generate python code, using multi-processing and socket
- could generate C, Java, and Erlang, too
- plan to generate PlusCal and others for verification

**optimization**:
- transform quantifications to aggregates & comprehensions
- build an interface to incrementalizer
- use InvTS for incrementalization
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
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.
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.
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)
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
1 class Coordinator extends Process:

2     def setup(tid, cohorts):
3         ncohorts = size(cohorts)  # number of cohorts
4         svoted = {}  # set of voted cohorts
5         nvoted = 0  # number of voted cohorts
6         sready = {}  # set of ready cohorts
7         nready = 0  # number of ready cohorts
8         sdone = {}  # set of done cohorts
9         ndone = 0  # number of done cohorts

10    def run():
11         send ('prepare',tid) to cohorts
12         await nvoted == ncohorts  # replaced universal quantification
13         if nready == ncohorts:  # replaced universal quantification
14             send ('commit',tid) to cohorts
15         await ndone == ncohorts  # replaced universal quantification
16         print('complete'+tid)
17     else:
18         s = sready  # replaced set query
19         send ('abort',tid) to s
20         print('terminate'+tid)
```python
# 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.