Distributed algorithms are at the core of distributed systems, which are increasingly indispensable in our daily lives. Yet, developing practical implementations of distributed algorithms with correctness and efficiency assurance remains a challenging, recurring task.

- Study of distributed algorithms has relied on either pseudocode with English, which is high-level but imprecise, or formal specification languages, which are precise but harder to understand, lacking mechanisms for building real distributed systems, or not executable at all.
- At the same time, programming of distributed systems has mainly been concerned with program efficiency and has relied mostly on the use of low-level or complex libraries and to a lesser extent on built-in mechanisms in restricted programming models.

What’s needed is (1) a simple and powerful language that can express distributed algorithms at a high level and yet has a clear semantics for precise execution as well as verification, and is fully integrated into widely used programming languages for building real distributed systems, together with (2) powerful optimizations that can transform high-level algorithm descriptions into efficient implementations.

1 A very high level language for clear description of distributed algorithms

We have developed DistAlgo, a very high-level language for expressing distributed algorithms that combines advantages of pseudocode, formal specification languages, and programming languages. We identified the following basic features needed for distributed algorithms, and designed DistAlgo to support them clearly and precisely:

(A) distributed processes that can send messages
(B) handling of received messages with support for atomicity
(C) waiting on conditions for synchronization, involving quantifications over sets of processes and history of messages sent and received, which are the deepest part of the algorithms
(D) configuration with library support

DistAlgo supports these features by building on an object-oriented language that supports high-level queries and quantifications over sets and sequences. The result is that distributed algorithms can be expressed clearly at a high level, almost exactly like pseudocode, but also precisely, like in formal specification languages, and be executed as part of real applications, as in programming languages.

For example, the two-phase commit protocol, in a completely executable program, can be specified in DistAlgo as follows, where received m from p is a shorthand for (m from p) in received, and # indicates the start of a comment.
The main reason that algorithms written in DistAlgo are almost like pseudocode is that complex 
synchronization conditions can be expressed using high-level quantifications over sets and sequences, 
including especially the history of messages sent and received. Expressing synchronization condi-
tions at such a high level allows the correctness of the algorithms to be proved much more easily. 
However, if executed straightforwardly, each quantifier will cause a linear factor in running time, 
and any use of the history of messages sent and received will cause space usage to be unbounded.

2 Powerful optimizations for generating efficient implementations

The main challenges in generating efficient implementations are higher-level control structures and 
data types. The most expensive features, by far, are synchronization conditions expressed using 
quantifications over complex objects and high-level data types, which are most often the set of 
processes and the history of messages.

To address this problem, quantifications over the history of messages sent to and received from 
other processes must be performed incrementally as messages are sent and received. There has been 
much previous research on incrementalizing expensive computations, for set languages, recursive 
functions, logic rules, and objects, but not for general quantifications. Also, previously studied 
general methods are for centralized programs, not distributed programs.

Our method is to automatically transform each send and receive clause in the program into 
an update to the sequence for message history, incrementally maintain the truth values of synchro-
nization conditions and necessary auxiliary values as the sequences of messages sent and received 
are updated, and finally remove those sequences as dead variables.

For example, for the two-phase commit protocol above, our method yields the following optim-
ized implementation for the body of class Coordinator; class Cohort and method main remain 
unchanged. Three universal quantifications and a set query are now incrementally maintained in 4 
counts and 3 sets, and each test of synchronization condition and each message handler takes only 
constant time. In particular, the history of unbounded message sequence received is removed, 
and the sets maintained are linear in the number of cohorts. This optimized program is much more 
tedious and error-prone than the original program, growing from 12 lines to 33 lines, and scattering 
the maintenance of the synchronization conditions and set queries in lower-level message handlers, 
which makes it much harder to prove the correctness of the algorithm.
To incrementally maintain the truth values of general quantifications, our method first transforms them into set queries. In general, however, translating nested quantifications simply into nested queries can incur asymptotically more space and time overhead than necessary. The key idea is to minimize the nesting of queries. This avoids maintaining unnecessary intermediate results, saving both time and space and yielding also simpler programs. Systematic incrementalization also allows the time and space complexity of the generated programs to be easily analyzed. Overall, our method allows efficient implementations to be generated, without manually coding the algorithms and applying ad hoc optimizations. It can even lead to simplification of the original algorithms.

3 Implementations and experiments

We have implemented a prototype of DistAlgo by extending the Python programming language, because Python has rich support for very high-level constructs for ease of programming, and simple and consistent syntax for ease of reading; Python is also widely used in practice and increasingly used in teaching.

We have applied the method to a set of well-known distributed algorithms, including different variants of algorithms for distributed mutual exclusion, leader election, atomic commit, and Paxos and Byzantine Paxos for distributed consensus.

We performed several sets of experiments, which helped show that (1) DistAlgo allows these algorithms to be expressed more easily and clearly, compared with other languages, (2) our optimizations improve the time and space asymptotically as analyzed, and (3) generated implementations, even though not yet optimized for constant factors, are not too far in performance from manually optimized programs. The figure above shows the running times of two-phase commit in DistAlgo before and after incrementalization, for failure rates of 0 (Commit in legend) and 100 (Abort in legend), averaged over 50 rounds and 15 independent runs, measured on an Intel Core-i7 2600K CPU with 16GB of main memory, running Linux 3.0.0 kernel and Python 3.2.2.

Directions for future work include formal semantics for the language, verification, additional optimizations, and improved implementations. A detailed description of this work is given in [Liu+11].