I Do Declare: Consensus in a Logic Language

Peter Alvaro, Tyson Condie, Neil Conway, Joseph M. Hellerstein, Russell Sears

June 5, 2009

Abstract

The Paxos consensus protocol can be specified concisely, but is notoriously difficult to implement in practice. We recount our experience building Paxos in Overlog, a distributed declarative programming language. We found that the Paxos algorithm is easily translated to declarative logic, in large part because the primitives used in consensus protocol specifications map directly to simple Overlog constructs such as aggregation and selection. We discuss the programming idioms that appear frequently in our implementation, and the applicability of declarative programming to related application domains.

1 Introduction

Consensus protocols are a common building block for fault-tolerant distributed systems [2]. Paxos is a widely-used consensus protocol, first described by Lamport [4, 5]. While Paxos is conceptually simple, practical implementations of the algorithm are notoriously difficult to achieve, and typically require thousands of lines of carefully written code [1, 3, 7].

Much of this implementation complexity arises because high-level protocol specifications must be translated into low-level imperative code, yielding a massive increase in program size and complexity. In practical implementations of Paxos, the simplicity of the consensus algorithm is obscured by common but often tricky implementation details such as event loops, timer interrupts, explicit concurrency, and the serialization and persistence of data structures.

By contrast, consensus protocols such as two-phase commit and Paxos are specified in the literature at a high level, in terms of messages, invariants, and state machine transitions. Overlog supports each of these concepts directly. By using a declarative language to implement consensus protocols, we hoped to achieve a more concise implementation that is conceptually closer to the original protocol specification. We discuss our Paxos implementation below, and describe how we mapped concepts from the Paxos literature into executable Overlog code.

Earlier work [11] implemented the Synod protocol (the kernel of Paxos) in P2 [6]. However, that work did not address practical details such as Multi-paxos, log replication, reconciliation, and leader election. Here, we describe a complete Paxos implementation that addresses these issues. More importantly, we reflect on the design patterns that we discovered while building this classical distributed service in a declarative language. The process of identifying these patterns helped us better understand why a declarative networking language is particularly well-suited to programming distributed systems. It has also clarified our thinking about the more general challenge of designing a more targeted domain-specific language for distributed computing.

1.1 Overlog

Overlog is a logic language based on Datalog. Datalog programs consist of rules that take the form:

```
head(A, B) :- clause1(A), clause2(B);
```

where `head`, `clause1`, and `clause2` are relations, "`:-`" denotes implication ($\Leftarrow$) and "`,`" denotes conjunction. A rule may have any number of clauses, but only a single head. The example rule ensures that the `head` relation contains a tuple (A, B) for each pair of A and B in the `clause` relations. A Datalog program begins with some base tuples, and derives new tuples by evaluating rules in a bottom-up fashion (substituting tuples in the clause relations to derive new tuples in the head relations) until no more derivations can be made. Such a computation is called a *fixpoint*. A set of rules essentially express the constraint that base facts and their transitive consequences will always be consistent at fixpoint.

Overlog computes a new fixpoint whenever new tuples arrive at a node. Overlog programs accept input from network events, timers, and native methods,

each of which produce new tuples. Because evaluation of an Overlog program proceeds in discrete time steps, rules may be interpreted as *invariants* over state: the consistency of the rule specifications will be true across all fixpoints.

Network communication is expressed using a simple extension to the Datalog syntax:

```
recv_msg(@A, Payload) :-
  send_msg(@B, Payload), peers(@B, A);
```

`@` introduces a *location specifier* field of a relation, and the associated variables A and B contain network addresses. A tuple moves between nodes if the address in its location specifier is distinct from the address of the node that deduced the tuple.

It is often useful to compute an aggregate on a set of tuples, typically to choose an element of the set with a particular property (e.g. min, max) or to compute a summary statistic over the set (e.g. count, sum). For example:

```
min_msg(min<SeqNum>) :-
  queued_msgs(SeqNum, _);
```

defines an aggregate relation that contains the queued message with the smallest sequence number, and

```
next_msg(Payload) :-
  queued_msgs(SeqNum, Payload),
  min_msg(SeqNum);
```

states that the content of `next_msg` is the payload of the queued message with the smallest sequence number. We encountered this pattern of *selection over aggregation* frequently when implementing consensus protocols.

Finally, Overlog allows special *timer* relations to be defined. The runtime inserts a tuple into each timer relation at a user-defined period; joining against a timer relation allows for periodic evaluation of a rule.

2 Two-phase commit

Before tackling a full-featured implementation of Paxos in Overlog, we began by using Overlog to build two-phase commit (2PC), a simple consensus protocol that decides on a series of Boolean values ("commit" or "abort"). Unlike Paxos, 2PC does not attempt to make progress in the face of node failures.

Both Paxos and 2PC are based on rounds of messaging and counting. In 2PC, the coordinator node communicates the state of a transaction to the peer nodes. When the transaction state transitions to "prepare" at a peer node, the peer responds with a "yes" or "no" vote. The coordinator counts these responses; if all peers respond "yes" then the transaction commits. Otherwise it aborts. In terms of the Overlog primitives described above, this is just messaging, followed by a `count` aggregate, and a selection for the string "no" in the peers' responses (Figure 1).

```
/* Count number of peers and "yes" votes */
peer_cnt(@Commander, count<Peer>) :-
  peers(@Commander, Peer);

yes_cnt(@Commander, Xact, count<Peer>) :-
  vote(@Commander, Xact, Peer, Vote),
  Vote == "yes";

/* Prepare => Commit if unanimous */
transaction(@Commander, Xact, "commit") :-
  peer_cnt(@Commander, NumPeers),
  yes_cnt(@Commander, Xact, NumYes),
  transaction(@Commander, Xact, State),
  NumPeers == NumYes, State == "prepare";

/* Prepare => Abort if any "no" votes */
transaction(@Commander, Xact, "abort") :-
  vote(@Commander, Xact, _, Vote),
  transaction(@Commander, Xact, State),
  Vote == "no", State == "prepare";

/* All peers know transaction state */
transaction(@Peer, Xact, State) :-
  peers(@Commander, Peer),
  transaction(@Commander, Xact, State);
```

Figure 1: Two-phase commit coordinator in Overlog. The first two columns of `transaction` are a primary key.

A practical 2PC implementation must address two additional details: timeouts and persistence. Timeouts allow the coordinator to return an error if the peers take too long to respond. This is straightforward to implement using timer relations (Figure 2).

Our Overlog implementation uses Stasis [10] to provide persistence on a per-table basis; depending on which variant of two-phase commit is in use (Presumed Commit, Presumed Abort, etc.), prepare, commit or abort messages should be persisted [8].

In short, the 2PC protocol is naturally specified in terms of aggregation, selection, messaging, timers, and persistence. Overlog provides each of these constructs, leading to an implementation whose size and complexity resemble the original pseudocode specification.

```
timer(ticker, 1000ms);

tick(Commander, Xact, C) :-
  transaction(Commander, Xact, State),
  State == "prepare", C := 0;

tick(Commander, Xact, C) :-
  ticker(),
  tick(Commander, Xact, Count),
  C := Count + 1;

transaction(Commander, Xact, "abort") :-
  tick(Commander, Xact, C), C > 10,
  transaction(Commander, Xact, State),
  State == "prepare";
```

Figure 2: Timeout-based abort. The first two columns of `tick` are a primary key.

2.1 Discussion

As we employed the primitives of messaging, timers and aggregation to implement 2PC, we found ourselves reasoning in terms of higher-level constructs that were more appropriate to the domain.

Multicast, a frequently occurring pattern in consensus protocols, can be implemented by composing the messaging primitive described in Section 1.1 with a join against a relation containing the membership list. The last rule in Figure 1 implements a multicast.

The `tick` relation introduced in Figure 2 implements a *sequence*, a single-row relation whose attribute values change over time. A sequence is defined by a base rule that initializes the counter attribute of interest, and an inductive rule that increments this attribute. Combining this pattern with timer relations allows an Overlog programmer to count the clock ticks, and therefore the number of seconds, that have elapsed since some event. This is the basis of our timeout mechanism (Figure 2).

A coordinator node can hold a *roll call* to discover which peers are alive by combining a coordinator-side multicast with a peer-side unicast response. A round of *voting* is a roll call with a selection at the peer (which vote to cast, probably implemented as selection over aggregation) and a `count` aggregate at the coordinator. The first three rules listed in Figure 1 implement this voting example of the roll call idiom.

Even in a simple case like 2PC, a variety of fairly typical distributed design patterns emerge quite naturally from the high-level Overlog specification. We now turn our attention to a more complicated protocol, Paxos, and see if these patterns are sufficient.

```
agent_cnt(Master, count<Agent>) :-
  parliament(Master, Agent);

promise_cnt(Master, Round, count<Agent>) :-
  send_promise(Master, Round, Agent, _);

quorum(Master, Round) :-
  agent_cnt(Master, NumAgents),
  promise_cnt(Master, Round, NumVotes),
  NumVotes > (NumAgents / 2);
```

Figure 3: We have quorum if we have collected promises from more than half of the agents.

```
can_promise(Agent, Round, OldRound, OldUpdate, Master) :-
  prepare(Agent, Round, Update, Master),
  prev_vote(Agent, OldRound, OldUpdate),
  Round >= OldRound;
```

Figure 4: An agent sends a constrained promise if it has voted for an update in a previous view.

3 Paxos and Multipaxos

In order to explore Paxos and its variants, we began by looking at the idealized protocol originally proposed by Lamport for reaching a single consensus. We then extended it to the Multipaxos protocol that can make an unbounded series of consensus decisions. In this section, we describe our Paxos implementation in terms of the idioms we identified for 2PC, and detail additional constructs that we found necessary.

3.1 Prepare Phase

Paxos is bootstrapped by the selection of a leader and an accompanying view number: this is called a *view change*. To initiate a view change, a would-be leader multicasts a *prepare* message to all agents; this message includes a view number that is globally unique and monotonically increasing. View numbers are implemented using a sequence that is seeded with the local network address and advanced by the number of agents in the view.

The Paxos protocol dictates that when an agent receives a *prepare* message, if it has already voted for a lower view number, it must respond with its previous vote. This *constrains* the update. Otherwise, it must send an unconstrained *promise* message. This invariant coupling requests and history is implemented with a query that joins the *prepare* stream with the local `prev_vote` relation (Figure 4). Finally, the prospective leader performs a `count` ag-

gregate over the *promise* messages; if it has received responses from a majority of agents then the new view has *quorum* (Figure 3). The prepare phase employs the idioms of sequences, multicast and counting.

3.2 Voting Phase

Once leadership has been established through this view change, the new leader performs a query to see if any responses constrain the update. If so, the leader chooses an update from one of the constraining responses (by convention, it uses a `max` aggregate over the view numbers). In the absence of constraining responses, it is free to choose an update from its request queue.

The remainder of the voting phase is a generalization of 2PC. The leader multicasts a *vote* message to all agents in the view, containing the current view number and the chosen update. Each agent joins this message against a local relation containing the agent's current view number. If the two agree, it responds with an *accept* message. An update is committed once it has been accepted by a quorum of agents; when the leader detects this, it responds to the client who initiated the update. The second phase of Lamport's original Paxos is a straightforward composition of multicast and counting.

3.3 Multipaxos

Multipaxos extends the algorithm described above to pass an ordered sequence of updates, and requires the introduction of additional state to capture the log history and the current instance number identifying the ordinal of the latest update. A practical implementation performs the *prepare* phase only once, and assuming a stable leader, carries out as many instances of the voting phase as are necessary.

Accommodating the notion of instances in our data model is a straightforward matter of schema modification. A *prepare* message now includes an *instance* number indicating the intended position of the next update in the globally ordered log. Each agent keeps track of the current instance number, and *promise* and *accept* message transmission is further constrained by joining against this relation: an agent only votes for a proposed update if its sequence number agrees with the current local high-water mark. Though the rule modifications to support instances and history were significant, we did not need to employ any new programming idioms.

```
top_of_queue(Agent, min<Id>) :-
  stored_update_request(Agent, _, _, Id);

begin_prepare(Agent, Update) :-
  stored_update_request(Agent, Update, _, Id),
  top_of_queue(Agent, Id);

delete
stored_update_request(Agent, Update, From, Id) :-
  stored_update_request(Agent, Update, From, Id),
  update_passed(Agent, _, _, Update, Id);
```

Figure 5: Choice and Atomic Dequeue.

3.4 Leader Election

Leader election protocols choose Multipaxos leaders, typically in response to leader failure.

When building a distributed file system on top of our Multipaxos implementation, we found that application semantics dictated that the client implement a particular set of timeout and policies. These policies led us to a simplistic, yet adequate leader election protocol. As an exercise, we also implemented the more general approach of [3] (which is presented in 31 lines of pseudocode) in 6 Overlog rules. The Overlog implementation was based on multicast, aggregation, sequences and timeouts, and left the Multipaxos implementation unchanged.

3.5 Discussion

Most of the logic of the basic Paxos algorithm is captured by combining voting with a sequence that allows us to distinguish new from expired views. Hence the idioms we described in our treatment of 2PC were nearly sufficient to express the significantly more complicated consensus protocol. Generalizing some of the more complicated implementation details, two new idioms emerged.

Using an exemplary aggregate function like `min` in combination with selection implements a *choice* construct that selects a particular tuple from a set. In Paxos, this construct is necessary for the leader's choice of a constrained update during the *prepare* phase. Combining the choice pattern with a conditional delete rule against the base relation allows us to express an *atomic dequeue* operation, which is useful for implementing data structures such as FIFOs, stacks, and priority queues. We found this construct useful as a flow control mechanism, to ensure that at most one tuple enters the *prepare* phase dataflow at a time (Figure 5).

4 Safety and Liveness

The correctness of a distributed system can be described in terms of its *safety* and *liveness* properties [9]. Informally, a safety property states that that nothing bad will happen during an execution of a system, while a liveness property states that something good will eventually happen. In other words, safety properties are invariants that ensure correctness of system state. Liveness properties ensure that forward progress is always made, and must be enforced using timeout mechanisms.

Overlog rules allow programmers to implement a distributed system in terms of their original specification: as a set of invariants, (e.g. [4]). In the case of 2PC the vote counting aggregate (which triggers once the agents unanimously vote "yes") is both the implementation of the protocol, and an invariant that enforces safety (Figure 1). Paxos depends on the invariant that a quorum is reached when more than half the agents have responded. This safety property is encoded as a choice by the last rule in Figure 3.

Overlog's ability to express timeouts allows us to easily specify liveness properties. Figure 2's timeout-based abort mechanism enforces a 2PC liveness invariant through counting and reference to physical time. Paxos' liveness is guaranteed by the leader election protocols discussed in Section 3.4.

Safety and liveness are typically specified in terms of three concepts: messaging, invariants and timeouts. Location specifiers define messages; selection over aggregation defines invariants, and sequences with timers define timeouts. An Overlog program specifies an implementation in terms of these idioms.

5 Conclusion

As the P2 authors discovered for network protocols [6], we found that a few simple Overlog idioms cover an impressive amount of the design space for consensus protocols. The correspondence between these idioms and consensus protocol specifications allows us to directly reason about the correctness of our implementations. We believe that such an approach will be fundamental to future approaches to distributed programming.

References

[1] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC*, 2007.

[2] M. J. Fischer. The consensus problem in unreliable distributed systems (A brief survey). In *Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, 1983.

[3] J. Kirsch and Y. Amir. Paxos for system builders. Technical Report CNDS-2008-2, Johns Hopkins University, 2008.

[4] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[5] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, December 2001.

[6] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev.*, 39(5):75–90, 2005.

[7] D. Mazières. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, January 2007.

[8] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM TODS*, 11(4):378–396, 1986.

[9] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, second edition, 1993.

[10] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, pages 29–44, 2006.

[11] Szekely, Benjamin and Torres, Elias. A Paxon evaluation of P2. `http://klinewoods.com/papers/p2paxos.pdf`.