I Do Declare: Consensus in a Logic Language[*]

Peter Alvaro   Tyson Condie   Neil Conway

Joseph M. Hellerstein   Russell Sears

{palvaro,tcondie,nrc,hellerstein,sears}@cs.berkeley.edu

UC Berkeley

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 [6, 7]. While Paxos is conceptually simple, practical implementations are difficult to achieve, and typically require thousands of lines of carefully written code [1, 4, 9].

Much of this implementation difficulty arises because high-level protocol specifications must be translated into low-level imperative code, yielding a significant increase in program size and complexity. In practical implementations of Paxos, the simplicity of the consensus algorithm is obscured by common but often tricky 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 pro-

tocols, 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.[1] 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 well-suited to programming distributed systems. It has also clarified our thinking about the more general challenge of designing a language for distributed computing.

Earlier literature described the modular decomposition of the Paxos protocol in terms of Timed I/O Automata [12]. Our approach differs in its grounding in database and logic languages rather than explicit representations of state machines, and in its aim to produce executable code rather than to model abstract systems. Szekely and Torres implemented the Synod protocol (the kernel of Paxos) in Overlog [14]. However, that work did not address important details such as Multipaxos, log replication, reconciliation, and leader election. Here, we describe a complete Paxos implementation that addresses these issues.

1.1 Overlog

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

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

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. Variables are denoted by identifiers that begin with an uppercase letter, or by the symbol "_", which indicates that the value of the variable will not be used in the rule. The example rule ensures that the `head` relation contains a tuple $\{A, C\}$ for each tuple $\{A, B\}$ in `clause1` and $\{B, C\}$ in `clause2` where the tuples have the same value for B. It does so by computing the join of `clause1` and `clause2` on B, and projecting A and C. 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 expresses the constraint

---

[*]Jim Gray observed about the two-phase commit protocol: "It is very similar to the wedding ceremony in which the minister asks 'Do you?' and the participants say 'I do' (or 'No way!') and then the minister says 'I now pronounce you,' or 'The deal is off.'" [3]

---

[1]The Overlog source code for the Paxos implementation we describe in this paper can be found at `http://db.cs.berkeley.edu/netdb-09/`.

```
/* Count number of peers */
peer_cnt(Coordinator, count<Peer>) :-
  peers(Coordinator, Peer);

/* Count number of "yes" votes */
yes_cnt(Coordinator, TxnId, count<Peer>) :-
  vote(Coordinator, TxnId, Peer, Vote),
  Vote == "yes";

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

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

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

Figure 1: 2PC coordinator protocol in Overlog. The DDL for `transaction` (not shown) specifies that the first two columns are a primary key.

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 may 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 at every fixpoint.

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

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

@ denotes the *location specifier* field of a relation, which indicates that 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 over 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 smallest sequence number among the queued messages, 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. This pair of rules is equivalent to the SQL statement:

```
/* Declare a timer that fires once per second */
timer(ticker, 1000ms);

/* Start counter when TxnId is in "prepare" state */
tick(Coordinator, TxnId, Count) :-
  transaction(Coordinator, TxnId, State),
  State == "prepare",
  Count := 0;

/* Increment counter every second */
tick(Coordinator, TxnId, NewCount) :-
  ticker(),
  tick(Coordinator, TxnId, Count),
  NewCount := Count + 1;

/* If not committed after 10 sec, abort TxnId */
transaction(Coordinator, TxnId, "abort") :-
  tick(Coordinator, TxnId, Count),
  transaction(Coordinator, TxnId, State),
  Count > 10, State == "prepare";
```

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

```
SELECT payload FROM queued_msgs
WHERE seqnum =
    (SELECT min(seqnum) FROM queued_msgs);
```

We encountered this pattern of selection over aggregation frequently when implementing consensus protocols.

Finally, Overlog allows special `timer` relations to be defined. The Overlog runtime inserts a tuple into each timer relation at a user-defined period, and the predicate holds only at these intervals. Thus, joining against a timer relation allows for periodic evaluation of a rule.

2. TWO-PHASE COMMIT

Before tackling Paxos, we used 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.

The mechanism that implements this protocol follows directly from the specification (Figure 1). The `peer_cnt` table contains the coordinator address and the number of peers. When `vote` messages arrive, the second and fourth rules are considered. If the fourth rule is satisfied (with a single "no" vote), the transaction state is updated to "abort"; otherwise, `yes_cnt` is incremented to reflect another positive vote for this transaction. If `yes_cnt` equals `peer_cnt`, the vote is unanimous and the transaction moves to the "commit" state. The fifth rule communicates changes to transaction state to every peer node.

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

```
promise(@Master, View, OldView, OldUpdate, Agent) :-
  prepare(@Agent, View, Update, Master),
  prev_vote(@Agent, OldView, OldUpdate),
  View >= OldView;
```

Figure 3: An agent sends a constrained promise if it has voted for an update in a previous view.

respond. This is straightforward to implement using timer relations (Figure 2). Our Overlog implementation uses Stasis [13] 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 [10].

In short, the 2PC protocol is naturally specified in terms of aggregation, selection, messaging, timers, and persistence. Focusing on these details led to an implementation whose size and complexity resemble the original pseudocode specification.

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. We call these higher-level constructs "idioms", and denote them with italics.

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 program to count the number of 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 and a set of peers can participate in a *roll call* to discover which peers are alive by combining a coordinator-side multicast with a peer-side unicast response. A `count` aggregate over a table containing network messages implements a *barrier*: the partial count of rows in the table increases with each received message, and synchronization is achieved when the count is high enough. 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 *barrier* at the coordinator. The first three rules listed in Figure 1 are an example of the voting idiom.

Even with a simple protocol like 2PC, a variety of common distributed design patterns emerge quite naturally from the high-level Overlog specification. We now turn to Paxos, a more complicated protocol, to see if these patterns remain sufficient.

3. PAXOS

Like 2PC, Paxos uses rounds of voting between a leader and a set of participating agents to decide on an update. Unlike 2PC, these roles are not fixed, but may change as a result of failures: this is central to Paxos' ability to make forward progress in the face of failures, even of the leader. We

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

promise_cnt(Master, View, count<Agent>) :-
  promise(Master, View, Agent, _);

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

Figure 4: We have quorum if we have collected promises from more than half of the agents.

began by implementing the Synod protocol, which reaches consensus on a single update, and then extended it to make an unbounded series of consensus decisions ("Multipaxos"). 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 *sequence* number that is globally unique and monotonically increasing.

The Paxos protocol dictates that when an agent receives a `prepare` message, if it has already voted for a lower view number, it must send a constrained `promise` message containing the update associated with its previous vote. Otherwise, it must send an unconstrained `promise` message, indicating that it is willing to pass any update the leader proposes. This invariant couples requests with history, and is implemented with a query that joins the `prepare` stream with the local `prev_vote` relation (Figure 3). Finally, the prospective leader performs a `count` aggregate over the set of `promise` messages; if it has received responses from a majority of agents then the new view has quorum (Figure 4). In sum, the prepare phase employs the idioms of *sequences, multicast* and *barriers*.

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 `promise` messages). In the absence of constraining responses, it is free to choose any pending update.

The remainder of the voting phase is a generalization of 2PC. The leader multicasts a `vote` message, containing the current view number and the chosen update, to all agents in the view. 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 *barriers*.

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

/* Select the enqueued update with the lowest Id */
begin_prepare(Agent, Update) :-
  stored_update_request(Agent, Update, _, Id),
  top_of_queue(Agent, Id);

/* Cleanup passed updates, causing top_of_queue
   to be refreshed */
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.

Rule Pattern	Idiom	Prepare	Propose	Election
All		13	13	19
Messages	Multicast	1	2	2
	Other	1	1	0
State Update	Sequence	2	2	3
	GC	1	3	2
	Other	0	1	6
Aggregation	Barrier	1	1	1
	Choice	2	1	0
	Other	5	2	3
Timer	Timeout	0	0	2

Figure 7: The usage of Overlog primitives and idioms in our Paxos implementation.

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 update history. A practical implementation performs the prepare phase once, and assuming a stable leader, carries out many instances of the voting phase.

Accommodating the notion of instances is a straightforward matter of schema modification. A prepare message now includes an instance number indicating the candidate position of the update in the globally ordered log. Each agent records 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.

3.4 Leader Election

Leader election protocols choose Multipaxos leaders, typically in response to leader failure. Detection of leader failure is usually implemented with timeouts: if no progress has been made for a certain period of time, the current leader is presumed to be down and a new leader is chosen. We implemented the leader election protocol of Kirsch and Amir [4] in 19 Overlog rules (the original specification required 31 lines of pseudocode). Our implementation was based on aggregation, *multicast*, *sequences* and *timeouts*, and left the core of our Multipaxos implementation unchanged.

However, this module is both the longest and most complicated component of the system. On reflection, our code resembled a superficial port of Kirsch and Amir's pseudocode, rather than a clean specification of invariants and transitions. In part, this is because leader election references a physical clock (to implement timeouts), unlike the rest of Paxos. The resulting Overlog code was overly mechanistic, consisting of a hodgepodge of timers, sequences, and backoff logic. We return to the difficulties of encoding liveness properties in Section 4.

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 this significantly more complicated consensus protocol. As we reflected on our implementation, 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).

Figure 6 illustrates the composition of the distributed programming idioms we encountered while implementing 2PC and Paxos. To avoid clutter, not every connection is drawn; for example, selection and join occur in nearly every idiom. Instead, we draw connections to emphasize interesting relationships: join combines with messaging to implement *multicast*, and selection and aggregation combine to implement *choice*. Unanimity, the critical safety property of 2PC, is enforced via a *vote* construct, as is the quorum constraint in both phases of Paxos. The safety of Paxos also relies on the invariant that an update accepted by any agent must be accepted by all: this is maintained by the *choice* of a constrained update in the prepare phase. Figure 7 shows the breakdown of our Paxos implementation in terms of Overlog primitives and higher-level idioms. Idioms like *multicast, barriers* and *sequences* occur ubiquitously, and aggregation is the most common rule pattern in all modules. "GC" denotes garbage collection rules that explicitly delete tuples that are no longer needed. As we would expect, timer logic is relegated to the time-aware leader election module.

4. SAFETY AND LIVENESS

The correctness of a distributed system can be described in terms of its *safety* and *liveness* properties [11]. Informally, a safety property asserts that "something bad never happens", while a liveness property states that "something good must eventually happen" [5]. In other words, safety properties are invariants that ensure correctness of system state. Liveness properties ensure that forward progress is always made, and are usually enforced using timeouts.

Overlog allows programmers to implement a distributed system as a set of invariants, which is often closer to its original specification (e.g. [6]). In the case of 2PC, the vote counting *barrier* (which triggers once the agents have unanimously voted "yes") is both the implementation of the protocol and an invariant that enforces safety (Figure 1). Paxos depends on the invariants that a quorum is reached
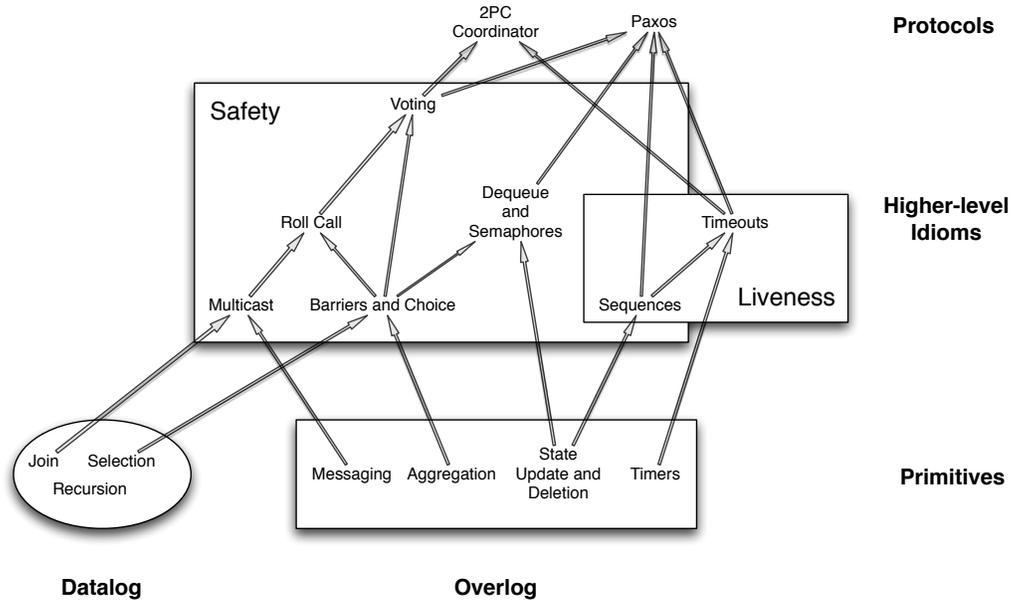
Protocols

2PC
Coordinator   Paxos

Safety   Voting

Dequeue
and   Timeouts
Roll Call   Semaphores

Higher-level
Idioms

Multicast   Barriers and Choice   Sequences   Liveness

Join   Selection
Recursion

Messaging   Aggregation   State
Update and   Timers
Deletion

Primitives

Datalog   **Overlog**

Figure 6: Distributed Logic Programming Idioms.

when more than half the agents have responded, and that all agents must accept an update accepted by any agent. These safety properties are encoded as a *barrier* by the last rule in Figure 4, and as a *choice* by the rule in Figure 3.

By contrast, liveness properties are not maintained by invariants that must hold at every timestep, but rather by restrictions that certain properties (for example, a pending request) hold only for a bounded amount of time. This requires considering infinite executions [11], which is not possible in Overlog. Instead, we are forced to use strategies similar to those employed in imperative languages, relying on timing to conservatively detect potentially infinite executions and take corrective measures.

Overlog's ability to reference real time via timers allows us to express these timing assumptions in protocols that can achieve liveness properties. Figure 2's timeout-based abort mechanism enforces a 2PC liveness invariant through *sequences* and reference to physical time. Paxos' liveness is achieved with a leader election protocol, as discussed in Section 3.4.

We found it much more difficult to reason about liveness properties in Overlog than to reason about safety properties. For example, convincing ourselves that a Paxos group will only accept an update if a majority of nodes have agreed is a simple matter of reading the *barrier* rule that defines a quorum, and observing that the accept rules are only reachable from a quorum. On the other hand, convincing ourselves that an update, once proposed, will eventually be accepted involves reasoning about timeout, retries, and potential livelock cycles between dueling proposers. This is less a criticism of the language than an observation about the difficulty of proving liveness properties in general. Although Overlog primitives such as timers and messaging allow the succinct expression of mechanisms for achieving liveness, Overlog lacks the ability to directly specify liveness properties as such. This results in a disconnect between abstract safety rules and lower-level timeout and retry logic. Hence, our Paxos implementation encodes safety properties declaratively, and liveness properties mechanistically.

5. CONCLUSION

When we set out to implement Paxos, our intent was to experiment with the generality of what was originally envisioned as a declarative networking language. As the P2 authors discovered for network protocols [8], we found that a few simple Overlog idioms cover an impressive amount of the design space for consensus protocols. The correspondence between Overlog idioms and consensus protocol specifications allowed us to directly reason about the correctness of our implementations.

In the course of our implementation, we saw significant reuse of higher-level constructs than those provided by Overlog. When comparing protocols, we found ourselves speaking in terms of *barriers*, *voting*, *choice* and *sequences* rather than select, project and join.

The use of these idioms made it easier to focus on the relatively small distinctions between protocol variants like 2PC, Synod and the Multipaxos variants, such as the conditions under which agents cast votes and when barriers may be passed. At a conceptual level, this highlights commonalities between these protocols that may deserve more attention; at a constructive level, it may provide guidance for the design of a library or domain-specific language for a larger class of classical distributed systems protocols.

The layering of idioms that we observed raised questions about the desired level of abstraction in future declarative languages. What "cut" in Figure 6 is best for what class of protocols? Is it best to provide a lower-level but general-purpose declarative language and layer libraries on top, or to design domain-specific declarative languages for only the trickiest aspects of distributed systems, leaving most tasks to other, more familiar languages? Perhaps most importantly, which approach is more likely to impact how developers build distributed systems?

Acknowledgments

6. REFERENCES

[1] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *PODC*, pages 398–407, 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*, pages 127–140, 1983.

[3] J. Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases*, pages 144–154, 1981.

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

[5] L. Lamport. "Sometime" is sometimes "not never": on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–185, 1980.

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

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

[8] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, 2005.

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

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

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

[12] R. D. Prisco, B. W. Lampson, and N. A. Lynch. Revisiting the paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 111–125, 1997.

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

[14] B. Szekely and E. Torres. A Paxon evaluation of P2. `http://klinewoods.com/papers/p2paxos.pdf`.