

A Paxos Evaluation of P2

Benjamin Szekely, Elias Torres

Harvard University

{bszekely,torres}@fas.harvard.edu

December, 2005

Abstract

P2 is a new and exciting system that permits declarative implementations of overlay networks. The recently published work cites reasonable implementations and performance for two protocols, Chord and Narada. Our work evaluates P2 through an implementation of a third protocol, the Paxos consensus protocol. P2's declarative language Overlog, admitted a smooth and readable implementation of the Paxos *preliminary protocol* [1]. Even though this implementation was a correct declaration of the Paxos *preliminary protocol*, P2's lack of concurrency control and well defined execution semantics provided a source difficult bugs and race conditions, and we remain uncertain as to whether or not our implementation is robust. Our initial conclusions are that Overlog is good language for implementing declarative overlays, but that P2 must provide more clear semantics and concurrency mechanisms for the system to become a widely adopted overlay network platform.

To evaluate the performance of P2, we built a simulator that could replay the P2 messages for various sized Paxos consensus instances, providing a lower bound on the time to reach consensus. We compared running times and CPU % utilization for identical conversations generated by executing P2 and by running our simulator called `p2replay`. Our preliminary experiments showed that P2 performed linearly with respect to the number of nodes, but was significantly slower and consumed much more of the CPU than the `p2replay` execution. For 256 nodes, P2 performed about 10 times as slow and consumed roughly 4 times as much of the CPU.

1 Introduction

In this paper, we describe the design, implementation and evaluation of the Paxos [1] consensus algorithm in the P2 declarative overlay network system [2]. P2 defines a declarative language called, OverLog, based on Prolog and Datalog to simplify the implementation and deployment of overlay networks. In particular OverLog can be used to implement standard distributed networking protocols and algorithms. The initial P2 research presents two example protocol implementations, each declared with an impressively small number of rules.

These algorithms were hand picked by the authors of the paper. Our research questions whether or not, other algorithms lend themselves to the same simple implementation and reasonable performance as Chord. We implement Paxos [1], a popular, distributed consensus algorithm with applications such as replication leader election in distributed database systems [3].

Paxos is a fairly complicated protocol and requires significant effort on the part of even the most accomplished computer scientist to fully grasp how and why it enables unreliable entities to reach consensus. Nevertheless, the protocol lends itself smoothly to implementation in Overlog because of its well de-

finer messages and node storage state. However, even though we were able to cleanly implement Paxos in Overlog, we found the P2 system to lack clean behavioral semantics. In this paper, we present, in detail, our implementation of Paxos in the Overlog language. In particular, we show how our implementation was guided by the operational semantics of P2.

To evaluate the performance of P2, we investigated the amount of total time spent in P2 (firing events, read/writing/joining tables, sending/receiving messages). To do this, we recorded the network conversation that the nodes carried out to reach consensus. Then, using a simulator we replayed the conversation and compared the running times of the simulated and actual consensus conversations. The difference is the amount of time spent in P2.

The remainder of the paper is organized as follows. In section 2 we discuss other declarative languages and Paxos implementations. For the benefit of the reader, we provide a brief description of the Paxos consensus protocol in section 3 and a brief overview of P2 and Overlog in section 4. In section 5 we present our implementation of Paxos in Overlog. In section 6 we describe our simulator and present our results. We discuss future work in section 7. In section 8 we reflect on our experience with P2 and Overlog. We conclude in section 9. We include a complete listing of our Paxos implementation in the Appendix A.

2 Related Work

As far as we know, we are embarking on the first attempt to implement and evaluate a distributed protocol in P2 other than those presented by the authors of the original paper [2]. Therefore we looked to the literature for Paxos implementations, benchmarking, and use cases to support our work. We found an extensive body of work regarding Lamport's original Paxos [1] paper in the form of improvements, slight variations or better explanations of the algorithm itself. Most of these papers were by Lamport, such as Paxos Made Simple [4], Fast Paxos [5] and Cheap Paxos. In contrast to the common feeling that Lamport's original metaphorical description of Paxos was prohibitively complex, we found it to be a very natural description of the protocol and used it for our main reference. Berkeley DB's replication master election algorithm [3] utilizes Paxos.

Unfortunately, very little work was found that tested the Paxos algorithm by itself, except for work by Hayashibara et. al. [6]. Hayashibara et. al. performed some experiments that

measure the performance of their implementation of two consensus algorithms: Paxos and Chandra-Toueg. We found their set of experiments a good starting point for our evaluation.

Obviously, we were unable to locate references to the P2 [2] work by Boon Thau Loo et. al. Nevertheless, we did a little bit of background reading of other work by the P2 group at Berkeley such as Declarative Routing [7].

3 Paxos

Paxos is not a simple protocol. Leslie Lamport's original paper [1] uses a metaphorical story about a make-believe ancient Greek island civilization's parliament. This initial description of the protocol was found to be too complex and so other papers and talks were created to provide a simpler explanation [4], [5]. Even these sources require a significant amount of time and concentration to comprehend and digest. Thus, we do not attempt to teach all of the subtleties of Paxos here. Instead, we will provide a brief overview of Paxos, highlighting several of its properties that guided our implementation. Lamport [1] describes several versions of Paxos. We focus on only what he calls the *preliminary protocol*.

Paxos is a consensus protocol. Nodes (represented as priests by Lamport) connected by a network (located in a parliament) use Paxos to agree on a single value (parliamentary decree). Any priest may propose a decree, vote (multiple times), or remain silent. The basic requirements of the *preliminary protocol* are that only a single decree is decided upon, and that this decree is voted for by a quorum of at least one more than half the priests. The protocol is designed in such a way that no wayward priest can break either of these rules. A dishonest priest can only inhibit progress; he cannot cheat and pass decrees that would otherwise fail. Even in the honest case, the preliminary protocol makes no guarantee about how fast a single decree is chosen.

A round of the protocol begins when a priest proposes a decree on a new ballot. We call a proposing priest P and a voting priest Q . The *ledger* is each priest's personal stable storage to record promises, votes and decrees. We now describe the steps in the protocol. For the motivation and theorems guiding the protocol, the reader should see Lamport's paper [1].

1. P sends a *nextBallot* message to all priests Q with decree dec_1 on ballot bal_1 . The purpose of this message is to see who would vote for such a decree before actually calling for a vote.
2. each Q who receives the *nextBallot* messages checks his records and responds to P saying either that he will be happy to vote for dec_1 , or that because of prior votes, he cannot, but that bal_2 with decree dec_2 is already in the works, so he would be happy to vote for dec_2 . The response is called a *LastVote* message. Q must record his response in his ledger before sending it out.
3. when P has received a *LastVote* message from a quorum ($\#(Q)/2 + 1$) of priests Q , he asks all priests Q in the quorum to vote on his ballot. If some priest in the quorum provided a *LastVote* message with an alternate decree, dec_2 ,

then P substitutes dec_2 for dec_1 on his ballot, asking the others priests to back this decree. The message sent to all priests Q in the quorum is called a *beginBallot* message.

4. when Q receives a *beginBallot* message for bal_1 , he must decide whether or to vote for its decree dec . Now Q 's other *LastVote* messages sent in step 2 (he may send many of these to different P 's), constitute promises not to vote on certain ballots. If Q has not made any such promises he can vote for dec on bal_1 , otherwise, he cannot vote. If he does decide to vote (he does not have to in either case), he sends a *VotedMessage* to P , and records the vote in his ledger.
5. when P has received a *Voted* message from every priest Q in the quorum for bal_1 , he writes down the decree in his ledger and declares success. He then sends a *Success* message to every priest Q .
6. when Q receives a *Success* message, he records the decree in his ledger.

We conclude this section with a few comments about Paxos. In steps 3 and 5 priest Q must count the number of such messages received including the current message, store the current message, and determine if enough messages have arrived to take further action. In sections 5 and 8, we will show how this basic function caused serious pitfalls due to false assumptions about P2's operational semantics.

P2 makes no distinction between stable and volatile storage so the following is only for interest. The promises, votes and decrees are the only data items that must be written to stable storage i.e., the ledger. If a promise or vote is forgotten by a priest, and he continues to participate, he may cause inconsistencies in the parliament, or greatly delay consensus. However, if a priest simply forgets that he is conducting a ballot (loses his quorum or vote count) then his decree will simply not pass and no rules will have been broken. Therefore, the priest may keep track of ballots on volatile storage which Lamport likens to slips of paper that the priests might easily lose.

4 P2 and Overlog

In this section we give a brief explanation of P2 and Overlog. For a complete description, the reader should see the original P2 paper [2]. We hope to describe enough here so that our implementation of Paxos in section 5 will make sense.

P2 is a system for building overlay networks using a declarative language called Overlog that is a derivative of Datalog and Prolog. Each node in a P2 overlay network stores data in relational database-like tables. Within a node, data flows through event streams. Data travels between nodes either through remote table accesses or through event streams. Rules, similar to database joins, define insertion of tuples into tables and the firing of event streams. Tables are defined through the *materialize* statement.

```
/* priest, priest */
materialize(priests, infinity, infinity, keys(2)).
```

The first argument is the name of the table. The second argument defines how many seconds a tuple remains in the table. The third argument specifies how many columns the table may have. The fourth argument declares which columns are keys. As far as we could tell, compound keys were not supported even though multiple columns could be keys. If a column declaration specified 2 and 3 as keys, then column 2 uniquely determines a row and column 3 uniquely determines a row, not the pair column 2 and column 3. The column definition of a table is implicitly defined by how the table is used. We found it to be a good convention to include a comment with the intended columns of the table. In this case, `priests` is a table that stores all the priests a particular priest knows about. We are not sure why, but it is important to include the local node (i.e. priest) as the first column in every table. Perhaps the authors intended to maintain a consistent unified view of a database table across all nodes.

The basic structure of an Overlog rule is `< ruleID >< head >: - < body >`. The `ruleID` is a unique identifier within the Overlog program. The `head` is either a table for insertion, or an event stream to fire. The `body` is a conjunction of tables, streams, conditions and variable assignments that determine how the table is populated or the event stream is fired. Dataflow is first initiated either by an event stream from another node or a local `periodic` event. Stream and table names can be appended with `@P` to indicate the node at which the stream or table should be populated or the node from which an event should arrive or at which a table should be looked up. The current node is implied but we always include it for clarity. The environment table is loaded from the command line arguments of the Overlog executable. The environment is way to provide external initialization input into the program. The environment comprises name-value pairs. Here, the names are numbers 1 to 1000 and the values are the other priests.

```
e0 priests@P(P,Q) :-
  periodic@P(P,E,0,1),
  env@P(P,H,Q),
  H in [1,1000].
```

Rule `e0` is evaluated when the built-in `periodic` event fires. In this case, we parameterize the periodic event to fire only once, with a delay of 0 seconds. This rule can most easily be understood as "When a periodic event fires at node P, for each value in the environment table for node P with a H name between 1 and 1000, store the corresponding priest Q in the priests table at node P". This rule only populates a table, and does not propagate the dataflow through a further event. The E parameter in the `periodic` event is a unique identifier for the event. The P2 paper does not provide an example of how to use E so we ignore it as well.

```
i0 sequence@P(P,I) :-
  periodic@P(P,E,0,1),
  I:=0.
```

Rule `i0` propagates the dataflow into the `sequence` event stream, with a value of 0 for I. When the `periodic` event fires, 0 is assigned to variable I. Then the `sequence` event stream is fired.

```
nb1 nextBallotMessage@Q(Q,P, BallotNo) :-
  prepareNextBallotEvent@P(P,_, BallotNo),
  priests@P(P,Q).
```

To fire an event stream on a remote node, we specify the remote node in the `head`. For example, Rule `nb1` reads "When we receive a `prepareNextBallotEvent` at node P indicating the start of the next ballot with ballot number `BallotNo`, send a `nextBallotMessage` with ballot number `BallotNo` to every priest Q in the `priests` table of P". This rule demonstrates the ease of multicast messaging. The evaluation of this rule effectively joins the current value of the `prepareNextBallotEvent` stream with the `priests` table in order to send a `nextBallotMessage` to every other priest. The `_` in the middle argument of the `prepareNextBallotEvent` indicates that this parameter is to be ignored. If the local node P is 1, and its priest table contains rows (1,2), (1,3) and (1,4), and a `prepareNextBallotEvent` (1, "mushrooms", 10) is fired, `nextballot` messages (2,1,10), (3,1,10), (4,1,10) are sent to priests 2, 3, and 4 respectively.

```
rnb1 maxVoteEvent@Q(Q,P, BallotNo, count<*>) :-
  nextBallotMessage@Q(Q,P, BallotNo),
  votes@Q(Q,PrevBallotNo, VoteDecree),
  PrevBallotNo < BallotNo.
```

Rule `rnb1` accepts the message, joins it with the `votes` table and does a comparison, firing the `maxVoteEvent` locally with the count of all matching tuples in the `votes` table. This rule counts all rows of the `votes` table with `PrevBallotNo` column value less than the `BallotNo` in the `nextBallotMessage`. `count<*>` indicates that only a single aggregation event should be fired with the count of all matching tuples. `count<*>` is a useful mechanism for testing existence in a table.

P2 makes no distinction between streams that flow within a node and streams that flow between nodes. However, we find it convenient to think of local streams as `Events` and inter-node streams as `Messages`. We name our streams accordingly. Rules may only have a single event stream in the `body`. That is, it is not possible to join on two event streams. However, the body may have as many variable assignments, conditional evaluations, and table lookups as needed.

We hope that by now, the reader has, if not enough to implement their own protocols, sufficient understanding to follow our implementation of Paxos. Before delving in, we introduce a concurrency problem with P2. Messages can arrive at a node at any time, causing tuples to enter the dataflow in pipeline fashion. Thus, we cannot assume that a node is processing only a single message at a time. While we are processing events fired by a first message, another message may come in and change the table state in a way that disrupts processing of the first message. P2 defines no operational semantics for concurrent messages and provides no mutex mechanism for sections of a dataflow. As a result, we had to carefully design our implementation to be robust enough to handle concurrent messages. We will discuss the problem further and suggest possible solutions in section 8.

5 Implementation

Before we begin, we note the brevity of our implementation. With 44 rules and 12 tables, we are able to implement a fairly complex distributed algorithm. We challenge the reader to implement Paxos in Java or C++ in fewer lines of code, without using a special purpose overlay package. The work of network communication and message marshalling/unmarshalling was hidden from us, and the relational table format was a natural way to store state in the Paxos protocol.

We now provide a detailed description of our implementation of Paxos in Overlog. It is not necessary to completely understand this discussion to understand our research. However, those interested in Paxos or Overlog may find our implementation decisions and programming style informative. A complete listing of the Paxos implementation can be found in Appendix A. We include detailed explanations of a few interesting rules that illustrate key strengths and weaknesses of P2.

Step 1 of the protocol is quite simple. A proposing priest sends a `nextBallotMessage` to every other priest in the parliament. This multicast is described in section ?? as rule `nb1`.

In Step 2 of the Paxos protocol, each priest Q checks his prior votes in the `votes` table. Rule `rnb1` computes a count of the number of votes on earlier ballots. `count<*>` is a very useful aggregation function for testing state. We often use `count<*>` to test and act upon existence of rows in a table. Rules `rnb2` and `rnb3` provide an example. Rule `rnb2` prepares a *LastVote* message in the case of at least one previous vote and `rnb3` prepares the message in the case of no previous vote. Though Prolog and Datalog purists may warn against overly complicated control flow, certain simple patterns can provide basic control. For example, `rnb2` and `rnb3` essentially define an if statement, or a branch in the dataflow graph.

```
rnb1 maxVoteEvent@Q(Q, P, BallotNo, count<*>) :-
  nextBallotMessage@Q(Q, P, BallotNo),
  votes@Q(Q,PrevBallotNo, VoteDecree),
  PrevBallotNo < BallotNo.
```

```
rnb2 prepareLastVoteEvent@Q(Q, P, BallotNo,
  VoteBallotNo, VoteDecree) :-
  maxVoteEvent@Q(Q, P, BallotNo, C),
  C == 0,
  VoteBallotNo := -1,
  VoteDecree := "<NULL>".
```

```
rnb3 prepareLastVoteEvent@Q(Q, P, BallotNo,
  max<VoteBallotNo>, VoteDecree) :-
  maxVoteEvent@Q(Q, P, BallotNo, C),
  C > 0,
  votes@Q(Q,VoteBallotNo, VoteDecree),
  VoteBallotNo < BallotNo.
```

In Step 3, we keep processing *LastVote* messages from the other priests Q until a quorum has been reached. The first two rules makes sure that we haven't received a message from the same priest twice. Paxos is designed to work in an environment where messages can repeat. As an aside, we would have liked to avoid rules like these using the key declaration of the tables.

However, we were unable to figure out deterministic and consistent behavior for tables with compound keys.

```
lv1 lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree, count<*>) :-
  lastVoteMessage@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree),
  ballotQuorum@P(P, BallotNo, Q).
```

```
lv2 newLastVoteMessageReceivedEvent@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree) :-
  lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree, C),
  C == 0.
```

When we fire an event signifying a new *LastVote* message, we immediately insert it into the `ballotQuorum` table. In Overlog, when only materialized tables appear in the *body* of a rule, the rule is evaluated whenever a tuple is inserted, sort of like an event on insertion. Rule `lv4` is evaluated when the tuple is inserted into `ballotQuorum`. For cleanliness, we fire a proper event to proceed. In our first initial implementation, we did not insert into the `ballotQuorum` table until the end of Step 3. We based our quorum calculations on the current state of the table (without the current message) plus the current message. We did this because we thought that a clean message processing technique was not to record state until the message was fully processed. Our hope was that P2 would process all events from a network message before firing events from another. However, when two messages came at once, events were fired concurrently, and we were unable to count the quorum properly because neither message had been inserted into the table. Inserting into the table right away seemed to correct the problem, though we remain cautious that further race conditions may yet occur. We elaborate on this issue in section 8

```
lv3 ballotQuorum@P(P, BallotNo, Q, VoteBallotNo,
  VoteDecree) :-
  newLastVoteMessageReceivedEvent@P(P, BallotNo,
  Q, VoteBallotNo, VoteDecree).
```

```
lv4 preCountBallotQuorumAfterInsert@P(P, BallotNo,
  VoteBallotNo, VoteDecree) :-
  ballotQuorum@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree).
```

Rules `lv61` - `lv63` serve two purposes. The first rule checks if, as of the most recent *LastVote* message, whether or not we have a quorum. The remaining rules, together, assure that we process a full quorum only once. That is, we might receive several *LastVote* messages after a quorum has been reached. However, we only want to initiate a vote once. Thus, the rules enforce that only a single *LastVote* message will lead to further processing, i.e. the first time a quorum is reached.

```
lv6 quorumReachedEvent@P(P, BallotNo,
  VoteBallotNo, VoteDecree) :-
  countBallotQuorumAfterInsert@P(P, BallotNo,
  VoteBallotNo, VoteDecree, C),
```

```

priestCount@P(P, PC),
C >= PC/2.

lv61 countBallotQuorumReachedEvent@P(P, BallotNo,
VoteBallotNo, VoteDecree, count<*>) :-
quorumReachedEvent@P(P, BallotNo, VoteBallotNo,
VoteDecree),
ballotQuorumReached@P(P, BallotNo, _, _).

lv62 ballotQuorumReached@P(P, BallotNo, VoteBallotNo,
VoteDecree) :-
countBallotQuorumReachedEvent@P(P, BallotNo,
VoteBallotNo, VoteDecree, C),
C == 0.

lv63 firstQuorumReachedEvent@P(P, BallotNo,
VoteBallotNo, VoteDecree) :-
ballotQuorumReached@P(P, BallotNo, VoteBallotNo,
VoteDecree).

```

In Step 4, when each priest Q receives a request to actually vote via a *beginBallotMessage* from P , Q must make sure that he has not sent any new *LastVote* messages that prevent him from voting on P 's ballot. This is the key to Paxos, and Lamport [1] explains it very well. If Q deems it acceptable to vote, he sends a *voted* message to P . We include this example because it shows how concisely one entire step in the protocol may be expressed.

```

rbv1 countConflictingPromisesEvent@Q(Q, P, BallotNo,
Decree, count<*>) :-
beginBallotMessage@Q(P, Q, BallotNo, Decree),
nextBallot@Q(Q, Ballot, PrevBallot),
BallotNo <= Ballot,
BallotNo >= PrevBallot.

rbv2 votes@Q(Q, BallotNo, Decree) :-
countConflictingPromisesEvent@Q(Q, P,
BallotNo, Decree, C),
C == 0.

rbv3 votedMessage@P(Q, P, BallotNo) :-
countConflictingPromisesEvent@Q(Q, P, BallotNo,
Decree, C),
C == 0.

```

In Step 5, Priest P must count up the votes so far and compare to the quorum each time a new vote arrives. Rules `cv4` - `cv7` add the vote to table of voters, count the votes, count quorum, then compare the two counts. This sequence of rules demonstrates how to properly compare the size of two tables. We perform an aggregation, and carry the value through a second aggregation, and then compare the two counts.

```

cv4 ballotVoters@P(P, BallotNo, Q) :-
newVoteEvent@P(P, BallotNo, Q).

cv5 countVotesEvent@P(P, BallotNo, count<*>) :-
ballotVoters@P(P, BallotNo, _).

```

```

cv6 countQuorumEvent@P(P, BallotNo, VC, count<*>) :-
countVotesEvent@P(P, BallotNo, VC),
ballotQuorum@P(P, BallotNo, _).

cv7 voteComplete@P(P, BallotNo) :-
countQuorumEvent@P(P, BallotNo, VC, QC),
VC == QC.

```

6 Evaluation

This section describes the experimental setup and performance results of our Paxos implementation in P2. Our goal for these preliminary experiments was to calculate the CPU utilization of a P2 node running the Paxos algorithm, and confirm or refute the hypothesis of the P2 paper that P2 provides servicable performance.

We do not attempt to measure bandwidth utilization or network latency. We believe that P2 uses a minimal wire format and that messages in our P2 implementation are comparable in size to messages in any native implementation of Paxos. Furthermore, P2 does not use any additional messaging to maintain connectivity or table state. We believe that CPU utilization will be where P2 incurs the most overhead.

6.1 Experimental Setup

Our experiments were run using a pre-release snapshot of the P2 libraries created on November 15, 2005. We built the libraries in several running Linux installations but due to limited administrative access and hardware availability, we were unsuccessful. Finally, we decided to use a clean build on a VMWare virtual machine instance of a Linux Debian distribution version 3.1r0a (a.k.a Sarge) with a 2.6.8 kernel image. The host machine was an Intel Pentium 4 CPU 2.8Ghz with 1GB RAM running Windows XP. The Linux installation was minimal and the only remaining packages left to install were gcc version 3.3.5 and sfs 0.8 (Self-Certifying File System), a major dependency in P2.

p2replay

Our main goal was to measure P2's per-node CPU utilization, but this measurement alone is of little value unless compared with alternative implementations in the same environment. However, we were unable to find a hand-coded Paxos implementation in a traditional language such as C++. It was suggested that we provide our own reference implementation by recording an actual run of our P2 implementation and replaying the conversation. While seemingly devious, the purpose of our simulation is to discover a lower bound on any highly-tuned implementation to estimate the P2 CPU overhead.

The main task of our implementation, *p2replay*, was to record the network conversation between two P2 nodes. We wrote a simple Overlog program that, using the "watch" built-in OverLog rule, logged tuples generated for either tables or streams. The watch output is logged to its respective node's output file. Problematically, this function logs inter-node tuples only on the receiving node and lacks source address information. Furthermore, the output is a human-readable version of the tuple and not the XDR ref binary wire format. Our solution modified two

Table 1: CPU utilization % for P2 and p2replay for different numbers of nodes. The utilization numbers are separated between parsing and execution.

# of nodes	4	8	16	32	64	128	256
p2replay							
parser	99.83	99.28	99.85	93.39	93.69	99.60	99.87
execution	21.68	17.61	16.91	16.57	16.64	14.12	14.69
p2							
parser	50.02	69.12	74.85	69.94	54.51	58.41	54.71
execution	1.67	2.80	6.25	15.45	36.35	62.86	73.36

Table 2: The total running times in seconds P2 and p2replay for different numbers of nodes. The running times are separated into parse times and execution times.

# of nodes	4	8	16	32	64	128	256
p2replay							
parser	0.007	0.014	0.028	0.058	0.115	0.135	0.248
execution	0.023	0.050	0.106	0.196	0.343	0.723	1.421
p2							
parser	0.196	0.163	0.124	0.145	0.222	0.155	0.194
execution	3.069	3.136	3.329	3.837	5.587	11.695	20.719

P2 dataflow element implementations to intercept messages before they were either sent or received. A few extra steps were necessary to make sure the proper unmarshalling was done and a hexadecimal dump of the tuple including source and destination address information was printed to the log file. A script then parsed the contents of the log generating a simplified replay script for each node.

To process the replay scripts, we implemented p2replay, a C++ UDP server program that executed the replay script. Upon start up it parses the script and loads a queue of messages, and exits when the queue is empty. After parsing, p2replay dequeues send-messages until it encounters its first receive-message. The server then blocks until it receives a message. Now, if the first message in the queue is a receive message, the contents are compared and if matched, the server scans and processes any remaining send elements until another receive element is found, then it blocks again. A highly probable case exists in which the messages arrive in a different order than the replay script expects. A simple fix is to search the queue for any receive message that matches the contents, process it and return to waiting status without processing any send messages in the queue.

We would like to point out that the P2 authors make note in their paper that their planner can be configured to connect a logging port for tuples sent over the network for debugging, but we were not able to find such a facility.

6.2 Results

Our current implementation supports only one proposing node but unlimited number of voting nodes. We generated two sets of

Table 3: The running times in system space in seconds P2 and p2replay for different numbers of nodes. The running times are separated into parse times and execution times.

# of nodes	4	8	16	32	64	128	256
p2replay							
parser	0.002	0.002	0.002	0.002	0.002	0.008	0.010
execution	0.004	0.006	0.012	0.022	0.035	0.089	0.187
p2							
parser	0.054	0.055	0.053	0.058	0.066	0.056	0.072
execution	0.050	0.084	0.198	0.569	1.947	7.206	14.686

Table 4: The running times in user space in seconds P2 and p2replay for different numbers of nodes. The running times are separated into parse times and execution times.

# of nodes	4	8	16	32	64	128	256
p2replay							
parser	0.005	0.012	0.026	0.052	0.105	0.127	0.238
execution	0.001	0.003	0.006	0.011	0.022	0.013	0.023
p2							
parser	0.035	0.037	0.033	0.037	0.039	0.019	0.022
execution	0.002	0.004	0.010	0.025	0.091	0.175	0.522

scripts that ran our two implementations (P2 and p2replay) 100 times for different numbers of voting nodes in powers of two up to 256 nodes. At first, we were interested in using more sophisticated profiling tools for our experiments. We toyed with oprofile, iperf, gprof, perfsuite, perfmon, etc. In the end, however, opted for basic Linux tools such as time and getrusage.

Both implementations perform a similar sequence of steps in a sample run: parsing of either the Overlog or replay script then beginning execution. We split our timings into two sections: parsing and execution. We calculated CPU time and utilization with gettimeofday and getrusage before and after each section. Next we subtracted start from stop times. Finally, we calculated CPU utilization. Timing the P2 binary proved difficult because it lacks a deterministic entry and exit point. Therefore, we inserted code into the dataflow elements that checked message flow for a specific *Success* message to a specific node to stop our timer. We suspect this to be the best choice, though a few extra messages could settle thereafter.

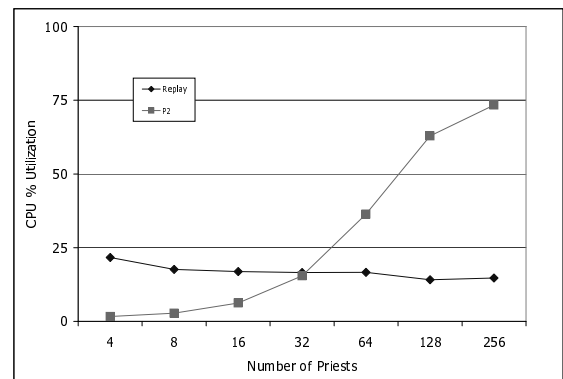


Figure 1: CPU Utilization

Our experiments ran when CPU load was at a minimum. We performed a code scan to disable any debugging and tracing flags in the P2 binaries and all remaining output was piped to /dev/null with the exception of the proposing node. We were unable to fully disable tracing in P2 to stderr, so we capture it in a file along with our timing reports. All network communication was performed through the local adapter. Network traffic was equivalent for both runs, with approximately a 3:2 message ratio for send and receive respectively growing linearly with the number of nodes (i.e. 189/126 for 64 nodes). We ran each exper-

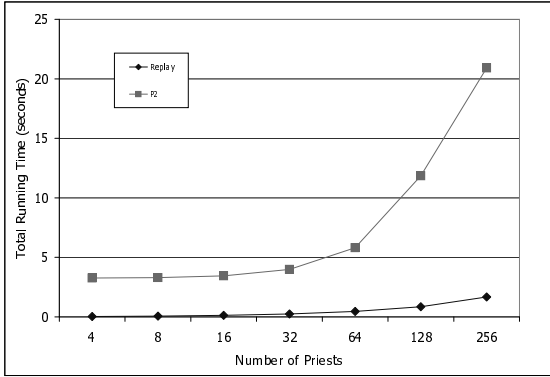


Figure 2: **Total Running Time**

iment 100 times and performed a standard average calculation for all runs without trimming our results.

Our numerical results are summarized in several tables. Table 1 shows CPU utilization across several runs of both algorithms, broken down between parsing and execution. Table 2 shows the total running time in seconds for both algorithms, again broken down into parsing and execution times. Tables 3 and 4 show the running times further broken down into system and user space respectively. A surprising feature of these tables is that parsing took more CPU but less overall time than execution for both p2replay and P2, though execution overtook parsing for P2 at 128 nodes. Another interesting result was that there was often a large discrepancy between the total clock time and the sum of the user and system space times. The difference is most likely spent waiting on disk and network I/O.

Figure 1 shows the execution CPU % utilization plotted over the different sized experiments. Figure 2 shows the total wall-clock runningtime across experiments for p2replay and p2. We discuss the results in the next subsection.

The scripts and data for our experiments can be found at <https://code.torrez.us/repos/public/2005/p2paxos/p2paxos-dist.tar.gz>

6.3 Discussion

After gathering our results from the different runs, nothing earth-shattering emerged from the numbers. As expected, parsing times showed a linear increase with the number of nodes for p2replay, but nearly constant for Paxos. The variance in overall wall clock time was expected since OverLog parsing and planning clearly requires additional processing than simply reading a replay script, line per line. Surprisingly, our 44 rule Paxos implementation (similar in size to P2’s sample 47 Chord rules), only took, on average, a fifth of a second from parsing to ready state. Bare in mind that overlay nodes should not have to be restarted often. Our results showed that P2 was not able to reach consensus in subsecond time. For example, a 4-node network took only 52ms of actual CPU time, but still needed almost 3 seconds from start to finish, not including process startup, pars-

ing and initialization. P2replay, on the other hand, was able to get subsecond total running time for all of its runs. These results were more inline with Hayashibara’s results of around 2ms.

Our main interest was CPU % utilization, which grew linearly with respect to the number of nodes. CPU % utilization doubled with the number of nodes, except on the 256 node experiment, where it only increased by 20%. We need to keep in mind that the amount of processing required to answer some of the queries is expected to grow with the number of messages sent across the network as lookups in soft-state tables grow more expensive with more data. The time spent in the kernel did grab our attention. We found that P2 was kernel-bound, and after profiling, we found that a large chunk of the time was spent manipulating I/O vectors in the suio helper class from the libasynch shared library. This makes sense, because although they are using a lot of reference counted objects in P2, there still is a lot of overhead during marshalling/unmarshalling. This process is done by creating a new Tuple structure, copying over all of the fields in the tuple, except the field being marshalled/unmarshalled, for which, a new chunk of memory must be allocated and the new data stored for the rest of the dataflow to complete. This could be an area where a pool of buffers could improve the performance, especially, since packet size is currently fixed to 1600 bytes.

Accidentally, we were forced to pay attention to memory usage when trying to run experiments with 128 priests or greater. The run was taking longer than expected to complete and after a short vmstat session, we noticed that the system was heavily swapping, since it had consumed our virtual image’s allocated 256MB of RAM. We increased the RAM to 768MB and we were able to execute our larger experiments. We were curious to find out where the memory usage was going, whether it was being used by storage in the proposing priest node or whether it was distributed evenly through the parliament. We are happy to report that it was evenly distributed and a single P2 node uses at least 2MB of RAM regardless of the usage.

Finally, we would like to reiterate that our results are only from one class of experiments in a rather awkward virtualized environment. We will be reporting more extensive results in future work and hopefully some more conclusive analysis of Paxos performance in P2.

7 Future Work

Thus far we have presented results and evaluation for a small number of experiments with our Paxos implementation in P2. Further exploration of the performance of P2 requires additional evaluation in a non-virtualized environment on different hardware. Running our experiments on multiple machines will offload the machine running the proposing priests as well as introduce more realistic network effects such as latency and packet loss. All nodes in our experiment, ran in the same machine, affecting the total running time for a proposal to reach agreement. It is not obvious how distributing the nodes across machines will affect running time.

We hope to fully implement the Paxos Complete Synod Protocol. We currently only have written the preliminary version of the protocol, which in fact has not been tested with multiple proposing priests. We believe most of the logic is already in

place, but new bugs will certainly appear as we test more complicated cases.

Even though we found Overlog quite natural for expressing Paxos, we encountered concurrency issues that merit further consideration. Specifically, we might try adding support for transactions and concurrency controls in the Overlog language and P2 planner. Such mechanisms would alleviate the programmer from having to do extra hacks to guarantee data integrity. In order for Overlog and P2 to be widely adopted as a method for building overlay networks, it must provide stronger semantics for concurrency, rule evaluation, and dataflow.

8 The P2 Experience

We began our work without access to the P2 binaries. However, thanks to Petros Maniatis et al, we eventually obtained and compiled the P2 system. In the interim we spent some time familiarizing ourselves with Prolog, Datalog and other rule-based programming systems such as DES [8]. Through our experience, we became intrigued with the idea of programming overlay networks in P2[2].

Datalog is primarily an alternative to database query languages such as SQL. Although never commercially successful, Datalog was touted for its clean semantics and recursive query definitions. Therefore, the P2 team showed great innovation in using a logic-based declarative query language as the basis for an overlay network system. The P2 developers made a significant enhancement to Datalog to adapt it to distributed systems programming. *Location specifiers*, as discussed in section ?? annotate the components of a rule with the node at which a stream or table should be considered [2]. This feature hides many of the networking difficulties with distributed programming. However, as we implemented Paxos, we encountered concurrency difficulties that suggest improvements to P2 and Overlog. In the remainder of this section we elaborate on these difficulties that we have touched on several times thus far in the paper. We conclude the section with a possible solution.

The Paxos preliminary protocol did not require overly complex rules or conditions. Rather, Paxos required the design of a flow of messages within a known group of nodes. Overlog rules may make heavy use of the network when querying for information, such as querying tables on remote nodes while processing the body of a rule. However, we opted for a cleaner message-based approach whereby data is transferred between nodes only through event streams.

Unlike traditional Datalog programs, Overlog programs cannot only infer data, but they can also materialize new tuples for future querying in soft-state tables. The interaction between network messages and table state is the source of concurrency issues that require great care to avoid. The following example illustrates the problem.

```

/* Receive a lastVote message from Q and count our
   quorum size so far */
lv1 lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree, count<*>) :-
   lastVoteMessage@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree),
   ballotQuorum@P(P, BallotNo, Q).

```

```

/* Check to see if we have reached quorum based on
   the count aggregate from lv1. We have an implicit
   +2 in the count due to the current message and
   the proposing priest */
lv2 beginBallotMessage@P(P, Q, BallotNo,
   VoteBallotNo, VoteDecree) :-
   lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree, C),
   priestCount@P(P, PC),
   C = PC/2 - 1,
   ballotsTried@P(P, BallotNo, decree, _, _),
   priests@P(P, Q).

/* Update the quorum table with an entry for this
   ballot and priest Q if we have not yet reached
   quorum */
lv3 ballotQuorum@Q(P, BallotNo, Q) :-
   lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree, C),
   priestCount@P(P, PC),
   C < PC/2 - 1.

```

The three rules above appear to properly implement a simplified version of step 3 of the protocol. In summary, the proposing priest awaits responses from 1 more than half the priests in the parliament (including himself) before calling for a vote on his ballot. For simplicity in this example, we always use the proposing priest's original decree and ignore the decrees in the LastVote messages from other priests in the quorum. In Rule lv1 we wait for a *LastVote* message from a priest *Q*. The processing node then counts the number of previous messages and checks whether or not we have reached a quorum. Rule lv2 begins the vote in the case that quorum has been reached, and rule lv3 updates the quorum table if quorum has not been reached.

Unfortunately, this approach is fraught with error. Logically, the *BallotQuorum* table should be updated as the final stage of processing a *LastVote* message as it is above. This is important because if we had permanent storage, as opposed to soft state tables, we should commit tuples to tables only once we have successfully completed the transaction. We discovered, however, that two *LastVoteMessages* may arrive simultaneously.

To understand this problem, consider a parliament of three priests, a proposer *P*, and voters *Q* and *R*. A quorum should be reached when either *Q* or *R* responds with a *LastVote* message because then we will have a quorum of 2 out of 3 (including the proposer). *P*'s *BallotQuorum* table is initially empty when *LastVote* messages arrive concurrently from *Q* and *R*. *P* evaluates lv1 twice, once for each message, and both with a count<*> of $C = 0$. $PC/2 - 1$ evaluates to $3/2 - 1 = 0$, and so lv2 is invoked twice, and two sets of *BeginBallotMessages* are erroneously sent out. Now, if the *LastVote* messages had been evaluated sequentially, the first (say from *Q*) would have caused an insertion in *BallotQuorum* and then *R*'s message would have computed a count of $C = 1$ and lv3 would be properly executed the second time.

The opposite, but equally damaging problem can occur when

there are more than three priests. All `LastVote` messages may arrive simultaneously, and because of the high `PriestCount`, `lv2` may *never* fire because the `BallotQuorum` table will always be empty when the count is aggregated by each concurrent message.

As a partial work-around, we perform the insertion into `BallotQuorum` as soon as we receive a message and before any comparisons of the table. Fortunately, when no events are in the body of a rule, it will be evaluated whenever a tuple is added to one of the materialized tables referenced in the body. This allows us to perform comparisons and process the message only after it has been recorded in the `BallotQuorum` table as shown below.

```

/* Receive a lastVote message from Q and count our
   quorum size so far */
lv1 lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree) :-
   lastVoteMessage@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree).

/* Record the message in the quorum table */
lv2 ballotQuorum@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree) :-
   lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree).

/* fire a proper event that can be joined with
   BallotQuorum table itself */
lv3 ballotQuorumInsertedEvent@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree) :-
   ballotQuorum@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree).

/* count the size of the update BallotQuorum
   table */
lv4 countBallotQuorumEvent@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree, count<*>) :-
   ballotQuorumInsertedEvent@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree),
   ballotQuorum@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree).

/* Check to see if we have reached quorum based on
   the count aggregate from lv1. We have an implicit
   +1 in the count due the proposing priest */
lv5 beginBallotMessage@P(P, Q, BallotNo,
   VoteBallotNo, VoteDecree) :-
   countBallotQuorumEvent@P(P, BallotNo, Q,
   VoteBallotNo, VoteDecree, C),
   priestCount@P(P, PC),
   C = PC/2,
   ballotsTried@P(P, BallotNo, decree, _, _),
   priests@P(P,Q).

```

This approach has a couple drawbacks. The first, is that because the entire contents of the `LastVote` message must be propagated through the table insertion event, we are forced to store this extra data in the `BallotQuorum` table even though it will never need to be stored! The second drawback is that

insertion is done at the beginning of message processing. In our small example, this is not a significant issue. However, in a longer message pipeline with various error conditions, such a restriction could lead to data inconsistency. The final drawback is that this version requires two additional rules and is much less clean and readable.

Furthermore, the write-immediately approach does not entirely solve the concurrency problem. Two concurrent `LastVote` messages may both be inserted concurrently. In such a case, the comparison in `lv5` will fail for both messages. We could change the `=` to a `>=` but then the comparison will succeed for both messages which is also wrong. To work around this, we had to create additional rules to prevent multiple messages from beginning ballots. Even with these additional rules (`lv6 - lv63` in Appendix A), our intuition is that we are making the race condition less and less likely, but not eliminating it with provably correct code.

We employed these tricks successfully in a couple of spots, but concluded that these problems could affect many algorithms in addition to Paxos. Any algorithm or protocol in which nodes aggregate results from multiple nodes and respond upon meeting a certain condition assuredly will face similar problems. Part of the problem is that rules by design cannot perform more than one task. For example, if we were able to aggregate results (`count < * >`) within a rule's body, we could have collapsed those two extra rules and many others into a single rule, possibly enforcing better execution semantics.

We tried remedying the problem by modifying P2 itself, but were unsuccessful. We observed the dataflow and processing queues to see if we could prioritize rules based local events over those based on network messages. We thought that if we could wait until all processing was completed locally and then process network messages we could resolve our problem. However, due to the push-pull design and absence of a global state of the P2 router and the de-multiplexing component, we had no way of knowing when the local processing was completed, unless of course we modified all routing nodes to provide processing status. These modification would break down for algorithms that involve intermediary external network events.

We now present a rough sketch of how concurrency controls might be expressed in Overlog and implemented in P2. Extra tokens or namespaces added to rules could direct the planning stage of the Overlog processor. For example, below we have the original (but abbreviated) set of rules for processing *LastVote* messages. A programmer should be able to define *atomic* blocks and specify those rules that start and end the block (Overlog rules are by definition unordered) to drive the query planner. Limitations on *atomic* blocks would include that only the *start* rule may listen to events outside of the block. In return P2 would guarantee that no other rules or block of rules will be evaluated until an *end* rule of the current *atomic* block had been evaluated. A possible implementation would be to store with each rule element a unique atomic block id and validate that all events in bodies belong to the same block. P2's existing router implementation is single threaded, so it should be easy to maintain *atomic* block execution state globally. While evaluating an *atomic* block, we would queue all inbound messages and events not destined to rules in the currently executing

atomic block. This approach would surely solve the our concurrent message problem, though additional concurrency problems likely exist that require further consideration.

```
atomic {

start lv1 lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
    VoteBallotNo, VoteDecree, count<*>) :-
    ...

end lv2 beginBallotMessage@P(P, Q, BallotNo,
    VoteBallotNo, VoteDecree) :-
    ...

end lv3 ballotQuorum@Q(P, BallotNo, Q) :-
    ...

}
```

Overall, P2 and Overlog surpassed our initial expectations of the suitability of a declarative logic query language for implementing distributed systems. Though we believe the P2 authors have nearly reached their goal of providing a clean, high level declarative language for overlay networks, we feel further work remains to that end. We strongly believe that if we enhanced P2 with our suggested *atomic* block it would be make P2 much more likely to become the tool of choice for programming overlay networks.

9 Conclusion

In this paper, we set out to explore P2 as a tool for implementing declarative network overlays using OverLog. We chose Paxos as our distributed algorithm due to its utility in larger distributed systems and production software packages. We then sought out to conduct experiments with a variable number of nodes in the network.

As our first result, we were able to complete our first draft implementation without having to make fundamental changes to our alpha version of P2. Learning OverLog was definitely not too difficult and the time saved, not dealing with socket programming, binary structure manipulation and overall design of a native library, was well spent in ensuring the correctness of our algorithm's implementation. Our implementation only took 44 OverLog rules, and of course, this was just our first try. We foresee that with some syntactic sugar, the code could still be made a little bit more compact.

Our experiment results, although somewhat inconclusive, give us good hope that P2's current unoptimized performance might be reasonable, depending on the system performance vs. deployment time requirements. P2's memory footprint is relatively small and depending on the amount of nodes in the distributed network CPU % utilization could be as high 70% for periods of 20 or fewer seconds.

Overall, we are quite optimistic and look forward to future work on P2, especially on the language itself, where many improvements could give OverLog a real edge over native implementations of distributed algorithms in overlay networks.

10 Acknowledgements

We would like to thank Petros Maniatis, et. al. for giving us early access to P2. Without it our work would have been much less interesting and would have required us to implement a complete stack to evaluate Paxos. Instead, we were to able to delve in to the core of P2 and validate much of the research and experience that he and his colleagues, have acquired in the past few years. We would also like to thank Chet Murthy (the distributed systems helpdesk), for his invaluable insight, patience and availability during late hours of work.

References

- [1] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, may 1998.
- [2] Boon Thau Loo and Petros Maniatis and Tyson Condie and Timothy Roscoe and Joseph M. Hellerstein and Ion Stoica. The part-time parliament. *ACM Transactions on Computer Systems*, may 1998.
- [3] Sleepycat. <http://www.sleepycat.com/products/db.shtml>.
- [4] Leslie Lamport. Paxos made simple. <http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf>.
- [5] Leslie Lamport. Fast paxos. Technical Report MSR-TR-2005-112, Microsoft Research, 2002.
- [6] Naohiro Hayashibara and Peter Urban. Performance comparison between the paxos and chandra-toueg consensus algorithms. Technical Report IC-2002-61, Ecole Polytechnique Federale de Lausanne, July 2005.
- [7] Boon Thau Loo and Petros Maniatis and Tyson Condie and Timothy Roscoe and Joseph M. Hellerstein and Ion Stoica. Declarative routing: Extensible routing with declarative queries. *SIGCOMM*, aug 2005.
- [8] Des: Datalog educational system. <http://www.fdi.ucm.es/profesor/fernan/DES/>.

Appendix A: Complete Paxos Implementation in Overlog

```
/* The environment table */
/* name, value */
materialize(env, infinity, infinity, keys(2,3)).

/* Store the value of an incrementing integer */
/* priest, seqNo */
materialize(sequence, infinity, 1, keys(2)).

/* The other priests in the parliament */
/* Initialized from environment */
/* priest, priest */
materialize(priests, infinity, infinity, keys(2)).

/* The total number of priests */
/* preist, priestCount */
materialize(priestCount, infinity, 1, keys(2)).

/* LEDGER: LastVote messages from step 2,
    must be checked when voting */
/* priest, ballotNo, voteBallotNo */
materialize(nextBallot, infinity, infinity, keys(2)).

/* LEDGER: The table of votes a priest has made */
```

```

/* priest, ballotNo, prevDecree */
materialize(votes, infinity, infinity, keys(2)).

/* PAPER: Ballot that a priest is keeping track of */
/* priest, ballotNo, decree,
   maxPrevVoteBallotNo, prevVoteDecree */
materialize(ballotsTried, infinity, infinity, keys(2,3)).

/* PAPER: Stores ballots that have reached quorum */
/* priest, ballotNo,
   maxPrevVoteBallotNo, prevVoteDecree */
materialize(ballotQuorumReached, infinity, infinity).

/* PAPER: Quorum of ballot that a
   priest is keeping track of */
/* priest, ballotNo, priest */
materialize(ballotQuorum, infinity, infinity).

/* PAPER: Votes of ballot that a
   priest is keeping track of */
/* priest, ballotNo, priest */
materialize(ballotVoters, infinity, infinity).

/* LEDGER: Decrees that have passed */
/* priest, decree */
materialize(decree, infinity, infinity, keys(2)).

/* Decrees that a priest wishes to propose */
/* Initialized from environment */
/* priest, decree, seqNo */
materialize(decreeQueue, infinity, infinity, keys(2)).

/* Initialization */

e0 priests@P(P,Q) :-
  periodic@P(P,E,0,1),
  env@P(P, H, Q),
  H in [1,1000].

e1 countPriestsEvent@P(P, count<*>) :-
  priests@P(P,_).

e2 priestCount@P(P, PCC) :-
  countPriestsEvent@P(P, PC),
  PCC := PC + 1. /* must include ourselves*/

e3 decreeQueue@P(P, D, I) :-
  periodic@P(P,E,0,1),
  env@P(P, H, D),
  H in [1001,2000],
  I := -1.

i0 sequence@P(P,I) :-
  periodic@P(P,E,0,1),
  I:=0.

i1 initiateNextBallotEvent@P(P) :-
  periodic@P(P, E, 3).

i2 prepareNextBallotEvent@P(P, Dec, NewSeq) :-
  initiateNextBallotEvent@P(P),
  decreeQueue@P(P, Dec, I),
  I == -1,
  sequence@P(P, CurSeq),

NewSeq := CurSeq + 1.

i3 decreeQueue@P(P, Dec, NewSeq) :-
  prepareNextBallotEvent@P(P, Dec, NewSeq).

i4 sequence@P(P, NewSeq) :-
  prepareNextBallotEvent@P(P, _, NewSeq).

i5 ballotsTried@P(P, NewSeq, Dec,
  MaxPrevVoteBallotNo, PrevVoteDecree) :-
  prepareNextBallotEvent@P(P, Dec, NewSeq),
  MaxPrevVoteBallotNo := -1,
  PrevVoteDecree := "<NULL>".

/* Step 1 - Priest P send nextBallot
   message to all priests */
nbl nextBallotMessage@Q(Q, P, BallotNo) :-
  prepareNextBallotEvent@P(P, _, BallotNo),
  priests@P(P, Q).

/* Step 2 - Priest Q reply to
   nextBallot with lastVote */
rnb1 maxVoteEvent@Q(Q, P, BallotNo, count<*>) :-
  nextBallotMessage@Q(Q, P, BallotNo),
  votes@Q(Q,PrevBallotNo, VoteDecree),
  PrevBallotNo < BallotNo.

rnb2 prepareLastVoteEvent@Q(Q, P, BallotNo,
  VoteBallotNo, VoteDecree) :-
  maxVoteEvent@Q(Q, P, BallotNo, C),
  C == 0,
  VoteBallotNo := -1,
  VoteDecree := "<NULL>".

rnb3 prepareLastVoteEvent@Q(Q, P, BallotNo,
  max<VoteBallotNo>, VoteDecree) :-
  maxVoteEvent@Q(Q, P, BallotNo, C),
  C > 0,
  votes@Q(Q,VoteBallotNo, VoteDecree),
  VoteBallotNo < BallotNo.

rnb4 lastVoteMessage@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree) :-
  prepareLastVoteEvent@Q(Q, P, BallotNo,
  VoteBallotNo, VoteDecree).

rnb5 nextBallot@Q(Q, BallotNo, PrevBallotNo) :-
  prepareLastVoteEvent@Q(Q, P, BallotNo,
  PrevBallotNo, PrevVoteDecree).

/* Step 3 - Priest P count quorum and begin vote */

lv1 lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree, count<*>) :-
  lastVoteMessage@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree),
  ballotQuorum@P(P, BallotNo, Q).

lv2 newLastVoteMessageReceivedEvent@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree) :-
  lastVoteMessageReceivedEvent@P(P, BallotNo, Q,
  VoteBallotNo, VoteDecree, C),
  C == 0.

```

```

lv3 ballotQuorum@P(P, BallotNo, Q, VoteBallotNo,
  VoteDecree) :-
  newLastVoteMessageReceivedEvent@P(P, BallotNo,
    Q, VoteBallotNo, VoteDecree).

lv4 preCountBallotQuorumAfterInsert@P(P, BallotNo,
  VoteBallotNo, VoteDecree) :-
  ballotQuorum@P(P, BallotNo, Q,
    VoteBallotNo, VoteDecree).

lv5 countBallotQuorumAfterInsert@P(P, BallotNo,
  VoteBallotNo, VoteDecree, count<*>) :-
  preCountBallotQuorumAfterInsert@P(P, BallotNo,
    VoteBallotNo, VoteDecree),
    ballotQuorum@P(P, BallotNo, _, _, _).

lv6 quorumReachedEvent@P(P, BallotNo,
  VoteBallotNo, VoteDecree) :-
  countBallotQuorumAfterInsert@P(P, BallotNo,
    VoteBallotNo, VoteDecree, C),
  priestCount@P(P, PC),
  C >= PC/2.

lv61 countBallotQuorumReachedEvent@P(P, BallotNo,
  VoteBallotNo, VoteDecree, count<*>) :-
  quorumReachedEvent@P(P, BallotNo, VoteBallotNo,
    VoteDecree),
  ballotQuorumReached@P(P, BallotNo, _, _).

lv62 ballotQuorumReached@P(P, BallotNo,
  VoteBallotNo, VoteDecree) :-
  countBallotQuorumReachedEvent@P(P, BallotNo,
    VoteBallotNo, VoteDecree, C),
  C == 0.

lv7 prepareBeginBallotEvent@P(P, BallotNo,
  MaxPrevVoteBallotNo, MaxPrevVoteDecree) :-
  ballotQuorumReached@P(P, BallotNo,
    VoteBallotNo, VoteDecree),
  ballotsTried@P(P, BallotNo, _, MaxPrevVoteBallotNo,
    MaxPrevVoteDecree),
  MaxPrevVoteBallotNo > VoteBallotNo.

lv8 prepareBeginBallotEvent@P(P, BallotNo,
  VoteBallotNo, VoteDecree) :-
  useNextBallotTried2@P(P, BallotNo, VoteBallotNo,
    VoteDecree),
  ballotsTried@P(P, BallotNo, _, MaxPrevVoteBallotNo,
    MaxPrevVoteDecree),
  VoteBallotNo >= MaxPrevVoteBallotNo.

lv9 ballotsTried@P(P, BallotNo, Decree,
  VoteBallotNo, VoteDecree) :-
  prepareBeginBallotEvent@P(P, BallotNo,
    VoteBallotNo, VoteDecree),
  ballotsTried(P, BallotNo, Decree,
    MaxPrevVoteBallotNo, MaxPrevVoteDecree),
  VoteBallotNo > MaxPrevVoteBallotNo.

lv10 beginBallotMessage@Q(P, Q, BallotNo, Decree) :-
  prepareBeginBallotEvent@P(P, BallotNo,
    VoteBallotNo, VoteDecree),
  ballotsTried@P(P, BallotNo, Decree, _, _),
  VoteBallotNo == -1,
  priests@P(P, Q).

lv11 beginBallotMessage@Q(P, Q,
  BallotNo, VoteDecree) :-
  prepareBeginBallotEvent@P(P, BallotNo,
    VoteBallotNo, VoteDecree),
  VoteBallotNo != -1,
  priests@P(P, Q).

/* Step 4 - Priest Q decide to vote */
rbv1 countConflictingPromisesEvent@Q(Q, P, BallotNo,
  Decree, count<*>) :-
  beginBallotMessage@Q(P, Q, BallotNo, Decree),
  nextBallot@Q(Q, Ballot, PrevBallot),
  BallotNo <= Ballot,
  BallotNo >= PrevBallot.

rbv2 votes@Q(Q, BallotNo, Decree) :-
  countConflictingPromisesEvent@Q(Q, P, BallotNo,
    Decree, C),
  C == 0.

rbv3 votedMessage@P(Q, P, BallotNo) :-
  countConflictingPromisesEvent@Q(Q, P, BallotNo,
    Decree, C),
  C == 0.

/* Step 5 - Priest P count up votes,
  declare success */
cv1 voteInQuorumEvent@P(P, BallotNo, Q, count<*>) :-
  votedMessage@P(Q, P, BallotNo),
  ballotQuorum@P(P, BallotNo, QuorumMember, _, _),
  Q == QuorumMember.

cv2 newVoteInQuorumEvent@P(P, BallotNo, Q, count<*>) :-
  voteInQuorumEvent@P(P, BallotNo, Q, C),
  C > 0,
  ballotVoters@P(P, BallotNo, Q).

cv3 newVoteEvent@P(P, BallotNo, Q) :-
  newVoteInQuorumEvent@P(P, BallotNo, Q, C),
  C == 0.

cv4 ballotVoters@P(P, BallotNo, Q) :-
  newVoteEvent@P(P, BallotNo, Q).

cv5 countVotesEvent@P(P, BallotNo, count<*>) :-
  ballotVoters@P(P, BallotNo, _).

cv6 countQuorumEvent@P(P, BallotNo, VC, count<*>) :-
  countVotesEvent@P(P, BallotNo, VC),
  ballotQuorum@P(P, BallotNo, _).

cv7 voteComplete@P(P, BallotNo) :-
  countQuorumEvent@P(P, BallotNo, VC, QC),
  VC == QC.

cv8 decree@P(P, Decree) :-
  ballotsTried@P(P, BallotNo, Decree, _, _),
  voteComplete@P(P, BallotNo, Q).

cv9 successMessage@Q(Q, P, BallotNo, Decree) :-
  voteComplete@P(P, BallotNo),

```

```
priests@P(P,Q),  
ballotsTried@P(P, BallotNo, Decree, _, _).
```

```
/* Step 6 - Priest Q record passed decree */
```

```
rv1 decree@Q(Q, Decree) :-  
    successMessage@Q(Q, _, _, Decree).
```