I Do Declare

Peter Alvaro, Tyson Condie, Neil Conway, Joseph M. Hellerstein, Russell Sears
UC Berkeley
• Consensus protocols are hard to implement.
• Part of the problem is the gap between specification and implementation.
• An ideal language could express:
  – mechanisms (multicast, barriers, and voting)
  – properties (safety and liveness).
• We’ll share some lessons learned building 2PC and Paxos in a distributed logic language.
Outline

1. Consensus and commit
2. Implementing protocols in a logic language
3. Synchronization, asynchrony and failure
4. The protocols
5. Interposing consensus into an existing program
Why consensus?

• Background:
  – Time to build a Real System™ in a declarative language
  – First cut: BOOM Analytics
    • MapReduce + DFS, fully compatible with Hadoop, plus new features (http://bit.ly/hjJOu)

• Motivation 1 for consensus:
  – We needed it to realize our multi-master DFS implementation (improvement over HDFS/GFS)

• Motivation 2:
  – Good technical challenge

• Motivation 3:
  – It fills in the middle of our stack
The Commit and Consensus Problems

The common problem
• A group of computers need to reach a decision.
• Disagreement is unacceptable.
• Machines may fail by crashing.
• Correctness is evaluated in terms of safety and liveness properties.

The Distinctions
• Details!
## Commit and Consensus

<table>
<thead>
<tr>
<th>Commit</th>
<th>Consensus</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Database domain</strong></td>
<td><strong>Distributed systems domain</strong></td>
</tr>
<tr>
<td>Unanimous voting</td>
<td>Majority voting</td>
</tr>
<tr>
<td>Static master</td>
<td>Dynamic master</td>
</tr>
<tr>
<td>Master failure may defeat liveness</td>
<td>(one per ‘view’)</td>
</tr>
<tr>
<td>Realizable via a very simple protocol</td>
<td>Robust to n/2 – 1 failures</td>
</tr>
<tr>
<td></td>
<td>(including the master)</td>
</tr>
<tr>
<td></td>
<td>Solved by complex protocols and systems</td>
</tr>
<tr>
<td></td>
<td>Theory: Paxos, Viewstamped Replication, GCS.</td>
</tr>
<tr>
<td></td>
<td>Praxis: Chubby (Google), Zookeeper (Yahoo!)</td>
</tr>
</tbody>
</table>
Paxos

• Imagined in the 80s. Published in the 90s.
  – concurrent with Liskov’s equivalent *Viewstamped Replication*

• Theoretically simple. Notoriously complicated to build.

• WHY?
Lamport on Paxos

“In fact, it is among the simplest and most obvious of distributed algorithms.”

• Paxos Made Simple

“He [Butler Lampson] is the only person who understood the algorithm and its significance as soon as he saw it.

• Interview
Implementing protocols in a logic language
Datalog review

\[ \text{Datalog rule: } \text{parent}(A, Z), \text{parent}(B, Z), A \neq B; \]

\[ \text{New Datalog rule: } \text{sibling}(A, B) :\text{-} \]

Diagram:
- Project
- Join
- Select

\[ \text{sibling}(A, B) :\text{-} \]
\[ \text{parent}(A, Z), \text{parent}(B, Z), A \neq B; \]
Overlog: Datalog in time and space

Overlog extends Datalog with:
- Aggregate functions (count, max, etc)
- Networking
- State Update / Deletion
- Clocks
Overlog -- Messages

pong(@X, Y) :-
  ping(@Y, X);

If there is a tuple ping(Y, X) at location Y, there is a tuple pong(X, Y) at location X.
2PC and the marriage ceremony

“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.”

2PC and the marriage ceremony

Marriage is:

- A *contract*: the two parties are joined if and only if they both agree to be joined in the presence of a minister.
- A *protocol*: the minister asks both parties and collects responses. Only if both are affirmative is the ceremony successful.

It is easy to see that the protocol upholds the contract. Does the contract imply the protocol?
2PC and the marriage ceremony

Married(@Minister, Partner1, Partner2) :-
    i_do(@Partner1, Partner2, Minister),
    i_do(@Partner2, Partner1, Minister);

Is this a contract or a protocol?
Synchronization, asynchrony and failure
Overlog -- Messages

pong(@X, Y) :-
    ping(@Y, X);

Location Specifier
Multicast

mcast(@Host, Sender, Message) :-
  message(@Sender, Message),
  membership(@Sender, Host);

Composition of join and messaging.
Synchronizing

ping(@B, A) :-
  membership(@A, B),
  start_ping(@A);

pong(@A, B) :-
  ping(@B, A);

pong_cnt(A, count<@B>) :-
  pong(A, B);

After the start_ping() event, what does pong contain? What does pong_cnt contain?
ping(@B, A) :-
    membership(@A, B),
    start_ping(@A);

pong(@A, B) :-
    ping(@B, A);

pong_cnt(A, count<B>) :-
    pong(A, B);
Synchronizing -- Unanimity

ping(@B, A) :-
    membership(@A, B),
    start_ping(@A);

pong(@A, B) :-
    ping(@B, A);

pong_cnt(A, count<B>) :-
    pong(A, B);

member_cnt(A, count<B>) :-
    membership(A, B);

all_in(A) :-
    pong_cnt(A, Cnt),
    member_cnt(A, Cnt);

all_in() creates a *barrier* using the count aggregate
Synchronizing -- Majority

ping(@B, A) :-
    membership(@A, B),
    start_ping(@A);

pong(@A, B) :-
    ping(@B, A);

pong_cnt(A, count<B>) :-
    pong(A, B);

member_cnt(A, count<B>) :-
    membership(A, B);

enough_in(A) :-
    pong_cnt(A, PCnt),
    member_cnt(A, MCnt),
    PCnt > (MCnt / 2);

enough_in() can escape the barrier early
Two Phase Commit (2PC)

/* 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);
2PC – Timeouts and Sequences

/* 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";
/* 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";
Discovering the idioms

Datalog

Join  Selection
Recursion
Discovering the idioms

Datalog

Join  Selection  Recursion

Overlog

Messaging  Aggregation  State
Update and
Deletion  Timers
Discovering the idioms
Composing the idioms
Paxos vs 2PC

Paxos uses voting in two places:
• To “install” the reign of a new leader (View)
• To pass updates within a view

Sequences are key to Paxos
• A view number sequence and an update order sequence
• Critical to safety properties

Paxos votes are more complicated than Y/N:
• They may contain previously-accepted updates.
• There is no “N” vote, though an agent may ignore the request.
Paxos – a closer look
Paxos – a closer look

Sequence, multicast
Barrier, choice, Multicast, sequence
Voting
Multicast, barrier
Paxos – a closer look

Sequence, multicast

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);

Barrier, choice

Voting

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

accept_cnt(Me, View, SeqNo, count<Agent>) :-
accept(Me, Agent, View, SeqNo, _);

globally_ordered(Me, View, SeqNo, Update) :-
accept_cnt(Me, View, SeqNo, Cnt),
agent_cnt(Me, PCnt),
Cnt > (PCnt / 2),
send_propose(Me, _, View, SeqNo, Update);

Barrier
Paxos from the clouds

Spacetime

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>prepare</td>
<td>promise</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

Primary Source

1. Priest $p$ chooses a new ballot number $b$ greater than $\text{lastTried}[p]$, sets $\text{lastTried}[p]$ to $b$, and sends a $\text{NextBallot}(b)$ message to some set of priests.

2. Upon receipt of a $\text{NextBallot}(b)$ message from $p$ with $b > \text{nextBal}[q]$, priest $q$ sets $\text{nextBal}[q]$ to $b$ and sends a $\text{LastVote}(b, v)$ message to $p$, where $v$ equals $\text{prevVote}[q]$. (A $\text{NextBallot}(b)$ message is ignored if $b < \text{nextBal}[q]$.)

3. After receiving a $\text{LastVote}(b, v)$ message from every priest in some majority set $Q$, where $b = \text{lastTried}[p]$, priest $p$ initiates a new ballot with number $b$, quorum $Q$, and decree $d$, where $d$ is chosen to satisfy $B3$. He then sends a $\text{BeginBallot}(b, d)$ message to every priest in $Q$.

4. Upon receipt of a $\text{BeginBallot}(b, d)$ message with $b = \text{nextBal}[q]$, priest $q$ casts his vote in ballot number $b$, sets $\text{prevVote}[q]$ to this vote, and sends a $\text{Voted}(b, q)$ message to $p$. (A $\text{BeginBallot}(b, d)$ message is ignored if $b < \text{nextBal}[q]$.)

5. If $p$ has received a $\text{Voted}(b, q)$ message from every priest $q$ in $Q$, (the quorum for ballot number $b$), where $b > \text{lastTried}[p]$, then he writes $d$ (the decree of that ballot) in his ledger and sends a $\text{Success}(d)$ message to every priest.

Overlog

- $\text{lastTried}[\text{Priest},\text{Sock}] = \text{lastTried}[\text{Priest},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},\text{Decree},\text{Sock}] = \text{nextTried}[\text{Priest},\text{Sock},\text{Old}]
- \text{nextTried}[\text{Priest},\text{Sock},�
On the ground: Prepare phase

Public local_aru(Me, 0) :-
  paxos::self(Me),
  start(),
  notin global_history(Me, _, _, _);

Public prepare(@Him, Me, View, Aru) :-
  leader::leader(@Me, Leader, View),
  paxos::parliament(@Me, Him),
  local_aru(@Me, Aru),
  Leader == Me;

datalist(Agent, Master, View, Aru, SeqNo, Update, Type) :-
  global_history(Agent, SeqNo, _ Update),
  datalist(Agent, Master, View, Aru, -1, _ "Bottom"),
  SeqNo >= Aru,
  Type := "Ordered";

Public datalist(Agent, Master, View, Aru, SeqNo, Update, Type) :-
  accept(Agent, M, OldView, SeqNo, Update),
  datalist(Agent, Master, View, Aru, -1, _ "Bottom"),
  SeqNo >= Aru,
  Type := "Proposed";

datalist(Agent, Master, View, Aru, -1, "none", "Bottom") :-
  prepare(Agent, Master, View, Aru),
  leader::last_installed(Agent, LastView),
  LastView == View;

datalist_length(Agent, Aru, count<SeqNo>) :-
  datalist(Agent, _ _, Aru, SeqNo, _, _);

prepare_oklist(@Master, View, SeqNo, Update, Type, Len, Agent) :-
  datalist_length(@Agent, Aru, Len),
  datalist(@Agent, Master, View, Aru, SeqNo, Update, Type);

delete
  prepare_oklist(Master, View, SeqNo, Update, Type, Len, Agent) :-
    prepare_oklist(Master, View, SeqNo, Update, Type, Len, Agent),
    global_history(Master, SeqNo, _, Update);

prepare_oklist_cnt(Master, View, Agent, Len, count<SeqNo>) :-
  prepare_oklist(Master, View, Agent, Len, Cnt, Cnt2),
  Cnt == Cnt2;

quorum(Master, View) :-
  paxos::priestCnt(Master, PCnt),
  leader::leader(Master, Leader, View),
  Master == Leader,
  prepare_ok_cnt(Master, View, Agent, Len, count<Agent>),
  RCnt > (PCnt / 2);

global_history(Agent, SeqNo, Requestor, Update) :-
  prepare_oklist(Agent, View, SeqNo, Update, "Ordered", _, Requestor),
  notin global_history(Agent, SeqNo);
On the ground: Prepare phase

Public local_aru(Me, 0) :-
  paxos::self(Me),
  start(),
  notin global_history(Me, _, _, _);

Public prepare(@Him, Me, View, Aru) :-
  leader::leader(@Me, Leader, View),
  paxos::parliament(@Me, Him),
  local_aru(@Me, Aru),
  Leader == Me;

datalist(Agent, Master, View, Aru, SeqNo, Update, Type) :-
  global_history(Agent, SeqNo, _ Update),
  datalist(Agent, Master, View, Aru, -1, _ "Bottom"),
  SeqNo >= Aru,
  Type == "Ordered";

Public datalist(OldView, SeqNo, Update) :-
  accept(Agent, M, OldView, SeqNo, Update),
  datalist(Agent, Master, View, Aru, -1, _ "Bottom"),
  SeqNo >= Aru,
  Type == "Proposed";

data(Patient, Master, View, Aru, -1, "none", "Bottom") :-
  prepare(Patient, Master, View, Aru),
  leader::last_installed(Patient, LastView),
  LastView == View;

data_list_length(Master, Aru, count<SeqNo>) :-
  data_list_length(Master, Aru, SeqNo, Type);
On the ground: Propose phase

public max_proposal(Master, SeqNo, max<View>) :-
    paxos_prepare::prepare_oklist(Master, SeqNo, "Proposed", _, _);

public send_propose(@Agent, Master, MyView, Aru, Update) :-
    duty_cycle(@Master),
    paxos::parliament(@Master, Agent),
    max_proposal(@Master, SeqNo, View),
    paxos_prepare::quorum(@Master, MyView),
    paxos_prepare::local_aru(@Master, Aru),
    paxos_prepare::prepare_oklist(
        @Master, View, SeqNo, Update, "Proposed", _);
On the ground: Propose phase

public max_proposal(Master, SeqNo, max<View>) :-
paxos_prepare::prepare_oklist(Master, SeqNo, "Proposed", _);  

public send_propose(@Agent, Master, MyView, Aru, Update) :-
duty_cycle(Master),
paxos::parliament(@Master, Agent),
max_proposal(@Master, SeqNo, View),
paxos_prepare::quorum(@Master, MyView),
paxos_prepare::local_aru(@Master, Aru),
paxos_prepare::prepare_oklist(
    @Master, View, SeqNo, Update, "Proposed", _);  

duty_cycle(Me) :-
paxos::self(Me),
t();

send_propose(@Agent, Master, View, Aru, Update) :-
duty_cycle(Master),
paxos::parliament(@Master, Agent),
notin paxos_prepare::prepare_oklist(
    @Master, View, _, "Proposed", _),
paxos_prepare::quorum(@Master, View),
leader::last_installed(@Master, View),
paxos_prepare::local_aru(@Master, Aru),
q(@Master, Update, R, Id),
top_q(q(@Master, Id));

public top_q(Master, min<Id>) :-
q(Master, _, _, Id);

delete
q(Me, Update, Sender, Id) :-
q(Me, Update, Sender, Id),
globally_ordered(Me, _, Update);  

public paxos_prepare::accept(@Other, Agent, View, SeqNo, Update) :-
send_propose(@Agent, _, View, SeqNo, Update),
notin paxos_prepare::global_history(@Agent, SeqNo, _, Update),
paxos::parliament(@Agent, Other, _);  

public delete
paxos_prepare::accept(Agent, Master, View, SeqNo, Update) :-
paxos_prepare::accept(Agent, Master, View, SeqNo, Update),
View2 > View;

public delete
paxos_prepare::accept(Agent, Master, View, SeqNo, Update) :-
paxos_prepare::accept(Agent, Master, View, SeqNo, Update),
globally_ordered(Agent, _, SeqNo, Update);

public accept_cnt(Me, View, SeqNo, count<Agent>) :-
paxos_prepare::accept(Me, Agent, View, SeqNo, _);
globally_ordered(Me, View, SeqNo, Update)
    :-
accept_cnt(Me, View, SeqNo, Cnt),
paxos::priestCnt(Me, PCnt),
Cnt > (PCnt / 2),
send_propose(Me, _, View, SeqNo, Update);

leader::progress_timer(Me, Start, Duration) :-
leader::progress_timer(Me, _, _),
leader::progress_timer_start(Duration),
paxos_prepare::global_history(Me, _, _),
Start := System.currentTimeMillis();

public hmax(Agent, max<SeqNo>) :-
paxos_prepare::global_history(Agent, SeqNo, _);

public paxos_prepare::local_aru(Agent, SeqNo + 1) :-
hmax(Agent, SeqNo);

public paxos_prepare::global_history(Agent, SeqNo, Requestor, Update) :-
globally_ordered(Agent, View, SeqNo, Update),
Requestor := "?";

public delete
paxos_prepare::prepare_oklist(Master, SeqNo, Update, Type, Len, Agent) :-
paxos_prepare::prepare_oklist(Master, SeqNo, Update, Type, Len, Agent),
globally_ordered(Master, GVView, GSeqNo, Update);
Interposing consensus into an existing program
BOOMFS mkdir()
BOOMFS mkdir()

pending(@Master, Parent, DirName, "mkdir") :-
    request(@Master, _, "mkdir", {Parent, DirName});

file(@Master, FileId, DirName, FParentId, true) :-
    passed(@Master, Parent, DirName, "mkdir"),
    fqpath(@Master, Parent, FParentId),
    FileId := newFileId();
Epilogue
Idioms

Datalog
- Join
- Selection
- Recursion

Overlog
- Messaging
- Aggregation
- State Update and Deletion
- Timers

Barriers and Choice
- Roll Call
- Voting

Multicast
- Timeouts
- Sequences
Properties

- Proper Des Timers Messaging Aggregation State Update and Deletion
- Join Selection Multicast Barriers and Choice Roll Call Voting
- Dequeue and Semaphores Sequences Timeouts
- 2PC Coordinator Paxos
- Safety
- Timeouts Dequeue and Semaphores
- Multicast Barriers and Choice
- Messaging Aggregation State Update and Deletion Timers
- Datalog
- Overlog
- Join Selection Recursion
What is the right cut?

Plain old Datalog

Overlog adds...

Selection  Join  Recursion

Multicast  Counting, Choice

Voting

Roll Call

Multicast

Deque and Semaphores

Sequences

Timeouts

Dequeue and Deletion

State Update and Deletion

Timers

Counting, Choice

Plain old Datalog
Conclusion

• We implemented 2PC and Paxos in Overlog.
• We developed a vocabulary of idioms for building these protocols.
• We discovered a clean mapping from logical constructs to correctness properties.
• Future work: extending the language with clean time semantics
Queries?