Verification of Implementations of Distributed Systems under Churn

Ryan Doenges, James R. Wilcox, Doug Woos, Zachary Tatlock, and Karl Palmskog
We should verify implementations of distributed systems...
...and we have!

<table>
<thead>
<tr>
<th>Framework</th>
<th>Prover</th>
<th>Verified system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verdi</td>
<td>Coq</td>
<td>Raft consensus</td>
</tr>
<tr>
<td>IronFleet</td>
<td>Dafny</td>
<td>Paxos consensus</td>
</tr>
<tr>
<td>EventML</td>
<td>NuPRL</td>
<td>Paxos consensus</td>
</tr>
<tr>
<td>Chapar</td>
<td>Coq</td>
<td>Key-value stores</td>
</tr>
</tbody>
</table>
...and we have!

<table>
<thead>
<tr>
<th>Framework</th>
<th>Prover</th>
<th>Verified system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verdi</td>
<td>Coq</td>
<td>Raft consensus</td>
</tr>
<tr>
<td>IronFleet</td>
<td>Dafny</td>
<td>Paxos consensus</td>
</tr>
<tr>
<td>EventML</td>
<td>NuPRL</td>
<td>Paxos consensus</td>
</tr>
<tr>
<td>Chapar</td>
<td>Coq</td>
<td>Key-value stores</td>
</tr>
<tr>
<td>Framework</td>
<td>Prover</td>
<td>Verified system</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Verdi</td>
<td>Coq</td>
<td>Raft consensus</td>
</tr>
<tr>
<td>IronFleet</td>
<td>Dafny</td>
<td>Paxos consensus</td>
</tr>
<tr>
<td>EventML</td>
<td>NuPRL</td>
<td>Paxos consensus</td>
</tr>
<tr>
<td>Chapar</td>
<td>Coq</td>
<td>Key-value stores</td>
</tr>
</tbody>
</table>
Churn = nodes joining & leaving a system at run time
Existing frameworks don't distinguish between knowing an address.
and knowing a node's address.
Under churn, systems depend on a "routing table"
But it can't be correct all of the time!
It can only be correct given enough time without churn: *punctuated safety*
Our contributions

1. First-class support for churn in Verdi
2. An approach to verifying punctuated safety
3. Ongoing case studies
   - Tree-aggregation protocol
   - Chord distributed hash table
Today

- The tree-aggregation protocol
- Churn in Verdi
- Proving punctuated safety
An example: counting nodes
These Pis live in Zach's office.
We need them for experiments.
They're subject to churn...
but they can count themselves!
Tree-aggregation: the idea

Combine distributed data into a single global measurement

Why not just ping every computer involved?

• No fixed list of nodes under churn
• The network may not be fully connected
• Can't handle large networks efficiently
Tree-aggregation: 2 protocols

1. Tree building: constructing a tree in the network

2. Data aggregation: moving data towards the root of the tree

Counting Pis is a very simple example. The protocol can aggregate more interesting data.
A network of nodes
Tree building: a root
Tree building: broadcasting levels
Tree building: broadcasting levels

- parent is least neighbor
- level is parent's + 1
Tree building: broadcasting levels

- Parent is the least neighbor.
- Level is the parent's level + 1.
Aggregation: pending counts
Aggregation: send pending to parent
Aggregation: send pending to parent
The root gets the total count
Handling churn: failures
Handling churn: failures
Handling churn: failures

The diagram illustrates a network topology with several nodes. The numbers (+1 or -1) indicate the status of these nodes. A node with a +1 signifies an active connection, while a -1 indicates a failure. The red 'X' marks a failed node, indicating it is no longer operational. The network's resilience and handling of failures are highlighted through these connections and statuses.
Handling churn: failures
Handling churn: failures
Handling churn: joins
Handling churn: joins
Handling churn: joins
Handling churn: joins
We can't finish counting during churn
We can't finish counting during churn
We can't finish counting during churn
Correctness (punctuated safety): Beginning from a state reachable under churn, given enough time without churn, the count at the root node becomes and remains correct
Roadmap

• The tree-aggregation protocol
• Churn in Verdi
• Proving punctuated safety
Roadmap

• The tree-aggregation protocol

• Churn in Verdi

• Proving punctuated safety
Verdi workflow

1. Write your system as event handlers
2. Verify it using our network semantics
3. Run it with the corresponding shim
Handlers change local state and send messages.

Definition result :=
    state * list (addr * msg).

- new state
- where to send it
- what to send
Existing event: delivery

Definition result :=
    state * list (addr * msg).

Definition recv_handler
    (dst : addr)
    (st  : state)
    (src : addr)
    (m   : msg)
: result := ...
New event: node start-up

Definition result :=
    state * list (addr * msg).

Definition init_handler
    (h : addr)
    (knowns : list addr)
: result := ...
Semantics: fixed networks

Record net :=
{ l failed_nodes : list addr;
  packets : addr -> addr -> list msg;
  state : addr -> state l }.

Inductive step : net -> net -> Prop :=
| Step_deliver : ...  
| Step_fail : ...
Semantics: fixed networks

Record net :=
  {l failed_nodes : list addr;
   packets : addr -> addr -> list msg;
   state : addr -> state l}.

Inductive step : net -> net -> Prop :=
  l Step_deliver : ...
  l Step_fail : ...
  l Step_fail : ...

probably Fin n
Semantics with churn

Record net :=
{l failed_nodes : list addr;
  nodes : list addr;
  packets : addr -> addr -> list msg;
  state : addr -> option state l}.

Inductive step : net -> net -> Prop :=
| Step_deliver : ...
| Step_fail : ...
| Step_init : ...

Semantics with churn
Now we can start verifying some properties of tree-aggregation!
The shim lets us run a system
We trust that the semantics describe the behavior of the shim and the network.
Roadmap

• The tree-aggregation protocol
• Churn in Verdi
• Proving punctuated safety
Roadmap

- The tree-aggregation protocol
- Churn in Verdi
- Proving punctuated safety
Churn forces safety violations

• Routing information can't be right all the time, and this typically violates top-level guarantees

• In the case of tree aggregation, any churn invalidates a correct total count
Detour: safety and liveness properties

Safety: nothing bad ever happens

Liveness: something good eventually happens
Safety and liveness properties

Define execution = infinite sequence of system states, ordered by step relation.

Then a safety property can be proved by examining only finite prefixes of an execution.

A liveness property cannot be disproved by examining finite prefixes of an execution.
We can prove safety properties with inductive invariants

A predicate $P$ on states is an inductive invariant when
Inductive invariants

A predicate \( P \) on states is an **inductive invariant** when

- \( P \) holds for the initial state

\[ \begin{align*}
\end{align*} \]
Inductive invariants

A predicate $P$ on states is an inductive invariant when
- $P$ holds for the initial state
- $P$ is preserved by the step
Inductive invariants

If $P$ implies our safety property, we've shown safety for all reachable states without needing to describe infinite executions in our Coq code!
..but "the root node eventually has a correct count" isn't a safety property!
Punctuated safety properties
Punctuated safety properties

Reachable under churn
Safety after churn stops
Punctuated safety properties

Reachable under churn (→)

Safety after churn stops (→)

"eventually"
Punctuated safety properties

Reachable under churn

Safety after churn stops

Diagram showing a network of nodes connected by arrows, indicating the flow and relationships between them.
We don't know how to prove this yet
We don't know how to prove this yet

It's a liveness argument, not a safety argument
We need a way to talk about infinite executions: liveness can't be proved with only finite traces.
Representing infinite executions in Coq

(* Infinite stream of terms in T *)
CoInductive infseq (T : Type) :=
  Cons : T -> infseq -> infseq.

(* Stream of system states connected by step *)
CoInductive execution :
  infseq (net * label) -> Prop :=
  Cons_exec : forall n n',
    step n n' ->
    execution (Cons n' s) ->
    lb_execution (Cons n (Cons n' s)).
Reasoning about executions: linear temporal logic (LTL)

Next $P$

Always $P$

Eventually $P$

...and much, much more!
LTL in Coq

Inductive eventually P : infseq T -> Prop :=
  | E0 : forall s,
    P s -> eventually P s
  | E_next : forall x s,
    eventually P s ->
    eventually P (Cons x s).

CoInductive always P : infseq T -> Prop :=
  | Always : forall s,
    P s ->
    always P (tl s) ->
    always P s.
InfSeqExt: LTL in Coq

• Extensions to a library by Deng & Monin for doing LTL over infinite (coinductive) streams of events

• Coq source code is on GitHub at DistributedComponents/InfSeqExt
We still can't prove correctness

What if messages from one node are indefinitely delayed while messages from another are still delivered?

Intuitively such an execution is "unfair" to the first node.

We have to assume a *fairness hypothesis*. 
Weak fairness: If an action is eventually always enabled, then it is always eventually taken.
Labels: turning steps into actions

SetParent $hp$

SendCount $hc$

SetCount $hc$
SetCount \( h \ c \) is enabled at this state
SetCount $hc$ is not taken in this execution, but SendCount $hc$ is taken.
Note: fairness has to be implemented and assumed

The shim could fail to handle messages fairly and prevent liveness

The network could delay packets and schedule delivery events unfairly
We can now state correctness for tree aggregation!

\[ \forall \text{ex } r, \]
\[ \text{reachable\_under\_churn (hd ex)} \rightarrow \]
\[ \text{execution churn\_free\_step ex} \rightarrow \]
\[ \text{connected (hd ex)} \rightarrow \]
\[ \text{weakly\_fair ex} \rightarrow \]
\[ \text{eventually (always} \]
\[ (\lambda \text{ex'} => \]
\[ \text{correct\_sum\_at\_root (hd ex'))} \]
\[ \text{ex} \]
Roadmap

• The tree-aggregation protocol
• Churn in Verdi
• Proving punctuated safety
Roadmap

- The tree-aggregation protocol
- Churn in Verdi
- Proving punctuated safety
Thanks!

We're on GitHub:

• uwplse/verdi

• DistributedComponents/verdi-aggregation

• DistributedComponents/InfSeqExt
Acknowledgements

Partially supported by the US National Science Foundation under grant CCF-1438982