Verification of Implementations of Distributed Systems Under Churn

Ryan Doenges James R. Wilcox
Doug Woos Zachary Tatlock
University of Washington, USA
rdoenges@uw.edu
{jrw12, dwoos, ztatlock}@cs.washington.edu

Karl Palmskog
University of Illinois at Urbana-Champaign, USA
palmskog@illinois.edu

1. Introduction

In order to provide high availability and operate in unreliable environments, many critical applications are implemented as distributed systems. Unfortunately, the need to handle packet drops, machine crashes, and churn—the spontaneous arrival and departure of nodes to and from the network—has made these systems difficult to implement correctly in practice. Recent work provides mechanisms to formally verify implementations that tolerate packet drops and machine crashes; however, no extant verification framework supports reasoning about churn. To address this challenge, we introduce support for reasoning about churn to the Coq-based Verdi framework.

Churn makes it difficult for systems to always uphold desired safety properties. Instead, systems promise to progress towards full safety when networks are quiescent, and do their best to fight entropy the rest of the time. We term this type of guarantee *punctuated safety*. Such properties are challenging to verify because their proofs involve both invariance arguments over system actions and well-foundedness arguments over infinite sequences of system states under fairness hypotheses.

2. Verdi Background

Verdi [2] is a framework for implementing and formally verifying distributed systems in Coq. In Verdi, a distributed system is defined as a finite set of nodes that communicate with each other by exchanging messages over a network. Nodes react to messages from the network and input from the outside world by invoking event handlers. To run systems in real networks, Verdi users extract their certified event handler code from Coq to OCaml and link it to a trusted OCaml shim that implements network primitives.

Verdi systems are proved correct with respect to a *network semantics* that encodes the system's assumptions about the behavior (or misbehavior) of the underlying network as a step relation on the state of the entire system. Safety properties are established by induction on that step relation.

Verdi introduced *verified system transformers* (VSTs) to separate fault-tolerance mechanisms from application logic. A VST maps a system implementation developed and verified with respect to a network semantics to a new system implementation that is formally guaranteed to behave analogously to the given system in a different (and possibly much more complex) network semantics.

The largest Verdi case study to date is the verification of the key safety properties of the Raft distributed consensus protocol [3]. The Raft proof makes heavy use of *ghost transformers*, VSTs from a network semantics to itself that add ghost state only available at verification time to the system while guaranteeing that behavior of the system is otherwise identical.

$$\dfrac{\begin{array}{c} to \in N \setminus F \qquad P[from,\ to] = m :: ms \\ \Sigma[to] = \sigma \qquad H_{net}(to,\ from,\ m,\ \sigma) = (\sigma',\ l) \\ P' = P[from,\ to \mapsto ms] \mathbin{+\!\!+} l \end{array}}{(N,\ F,\ \Sigma,\ P) \rightsquigarrow (N,\ F,\ \Sigma[to \mapsto \sigma'],\ P')}\ \textsc{Deliver}$$

$$\dfrac{\begin{array}{c} h \notin N \qquad H_{init}(h) = \sigma \\ P' = P \mathbin{+\!\!+} [(a,\ \mathsf{New}) \mid a \in N \setminus F,\ h \sim a] \end{array}}{(N,\ F,\ \Sigma,\ P) \rightsquigarrow (\{h\} \cup N,\ F,\ \Sigma[h \mapsto \sigma],\ P')}\ \textsc{Start}$$

$$\dfrac{h \in N \setminus F \qquad P' = P \mathbin{+\!\!+} [(a,\ \mathsf{Fail}) \mid a \in N \setminus F,\ h \sim a]}{(N,\ F,\ \Sigma,\ P) \rightsquigarrow (N,\ \{h\} \cup F,\ \Sigma,\ P')}\ \textsc{Fail}$$

Figure 1. Selected rules from the dynamic network semantics used to verify tree-aggregation protocols. A network is a 4-tuple $(N,\ F,\ \Sigma,\ P)$ where N is a set of node addresses, F is a set of failed nodes, Σ is a partial map from node addresses to a user-defined node state type, and P is a map from pairs $(from, to)$ of addresses to a list of pending messages for to originating at $from$. If a and b are addresses, $a \sim b$ holds if the node at a and the node at b are adjacent in the overlay network. The arrow $\mapsto$ denotes an update of a map. The operator $(\mathbin{+\!\!+})$ takes a network map P and a list of pairs (to, m) and returns a new network map P' identical to P but with all messages m appended to the channel from h to *to*, where h is understood from context.

3. Semantics of Networks Under Churn

Verdi's existing network semantics assume a fixed set of node names throughout the lifetime of each system. For example, Verdi's network semantics for static networks with crash-reboot failures tracks failed nodes in a set F and records the state of each node in a total map Σ from node names to state. Figure 1 illustrates key rules from our new semantics for dynamic networks under churn. In particular, our new semantics adds a set N to track nodes that have joined the network and turns Σ into a partial map defined only on N. We assume an overlay network defined by an irreflexive and symmetric relation on node names. Neighbors in the overlay network detect one another's arrivals and failures.

The DELIVER rule lets a live node take a message off of a readable channel, perform some action H_{net} that updates its state, and send any number of messages to other nodes. Messages are received in the order they are sent, since in the tree-based aggregation case study discussed below reasoning with ordered delivery has been more convenient than making causality arguments.

The START rule has an uninitialized node h join the network, obtain an initial state from H_{init}, and send a New message to each neighbor of h in the overlay.

The FAIL rule lets a live node h fail and sends a Fail message to each neighbor of h in the overlay. Thus, if we have eventual message delivery, we have eventual detection of all failures.

Encoding networks under churn. Verdi systems provide the types `name`, `msg`, and `data` of node addresses, messages, and node state respectively. We represent configurations used in our network semantics with the following Coq record type.

```
Record nw := { nwState: name -> option data;
 nwFailed: list name; nwNodes : list name;
 nwPackets: name -> name -> list msg }.
```

The semantics is encoded by an inductive step relation like the following that has one constructor for each rule of the semantics.

```
Inductive step_churn : nw -> nw -> Prop :=
Start: (*..*) | Fail: (*..*) | Deliver: (*..*).
```

4. Reasoning About Liveness in Verdi

Establishing punctuated safety for a system requires proving a *liveness* property of all executions of the system starting from arbitrary configurations reachable in a network under churn. However, these proofs are carried out in a *quiescent* network semantics that prohibits churn (e.g., the semantics in Figure 1 without the START and FAIL rules). To support reasoning about liveness in Verdi, we define executions as *infinite sequences* of states joined by labeled steps. These executions are subject to *fairness hypotheses* on labels.

Infinite sequences. We encode executions in Coq using an extended version of an existing library for reasoning about infinite sequences based on coinductive types [1]. Our extended library supports all of the standard linear temporal logic operators (`always`, `eventually`, etc.), and includes most standard operator equalities along with many other lemmas we found useful in temporal reasoning.

Fairness. Many interesting liveness properties cannot be proved without assuming some form of fairness between the actions taken by a system. In particular, most liveness proofs would be impossible if message delivery could be deferred indefinitely. We define fairness based on the labels of steps between states in executions. A label can be viewed as a handle on a particular kind of node action. While the label on a particular transition is determined by the node event handlers run in that transition, labels are elided at extraction time. A label l is *enabled* at a state when the system is able to take a step labeled with l starting from that state. Weak fairness on labels, the key fairness notion we use, rules out executions where l becomes enabled forever but never actually occurs.

To check the feasibility of our support for liveness reasoning in Verdi, we proved a liveness property of a lock service. The lock service manages a single shared lock in a network semantics without failures. We proved that any client who requests the lock eventually receives it. Together with an earlier safety proof of *mutually exclusive* lock access for the same system, the new liveness proof establishes full functional correctness of the lock service.

5. System Decomposition

Conventional approaches to distributed systems verification focus on establishing correctness for *models* which abstract away implementation details and usually involve only one protocol at a time. Real-world systems rely on many separate protocols running concurrently. Even when the correctness of each subsystem is verified in isolation, properties of the top-level system may be violated due to unintended interactions or erroneous assumptions. At the same time, the large size of a complete system is a hindrance to verification from scratch.

We suggest using a methodology based on VSTs in Verdi to lift established properties of components of decomposed systems to their analogues at the complete system level. Subsystems obtained from decompositions process a strict subset of the top-level system's messages and evolve in lock-step with the complete system. In Coq, decompositions are encoded as partial maps between Verdi system definitions. This allows both safety properties about reachable configurations and liveness properties about infinite executions to be lifted from the subsystem where they were proven to a top-level system. Successful lifting of a property requires showing that the decomposition partial map preserves the property and all of its hypotheses. For liveness properties, this usually requires showing that some form of fairness is preserved by the partial map.

6. In-Progress Case Studies

Tree-Based Aggregation. Gathering data from the environment over time is an important part of systems like network management and sensor networking. In-network aggregation avoids centralized data processing and lowers link congestion. In this approach, data is sent from *source* nodes to a *sink* node which, in the absence of churn, is expected to eventually hold the global aggregate. To minimize the number of hops from source nodes to the sink node, nodes assemble and maintain a *spanning tree* for the network.

We have encoded a churn-tolerant protocol for tree-based aggregation in Verdi that performs data aggregation and tree building independently. The aggregation component relies on failure detection to guarantee that no data is lost or added when node joins and failures occur. The tree-building component ensures that in a quiescent network nodes eventually build a minimal spanning tree rooted at the sink. The complete system relies on the properties of both aggregation and tree-building to guarantee that aggregates are sound, complete, and flow towards the sink.

Chord. Chord is a lookup protocol designed to serve as the substrate for a distributed hash table. To assign responsibility for lookups equally among nodes, Chord gives each node a unique identifier by hashing their addresses into a circular identifier space. A global ring structure is maintained by local state at each Chord node consisting of pointers to nodes that are nearby in the identifier space. In an *ideal ring* these pointers include all nearby nodes. Most of the time, some nodes will be skipped due to churn rendering pointer data invalid or incomplete. Correctness for Chord requires the following punctuated safety property: any Chord ring disrupted by churn will eventually become and remain ideal if it is given enough time without churn. The published specifications of Chord did not satisfy this property [4]. Zave has written a new specification for Chord which is accompanied by a semi-automated proof of punctuated safety [5]. We are working on an analogous proof for a Verdi implementation of Zave's new version of Chord.

References

[1] Y. Deng and J.-F. Monin. Verifying self-stabilizing population protocols with Coq. In *TASE 2009*, July 2009.

[2] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and verifying distributed systems. In *PLDI 2015*, June 2015.

[3] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *CPP 2016*, January 2016.

[4] P. Zave. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2), Apr. 2012.

[5] P. Zave. How to make Chord correct (using a stable base). *CoRR*, abs/1502.06461, 2015. URL `http://arxiv.org/abs/1502.06461`.