Aneris: A Logic for Node-Local, Modular Reasoning of Distributed Systems

MORTEN KROGH-JESPERSEN, Aarhus University, Denmark
AMIN TIMANY, imec-Distrinet, KU Leuven, Belgium
MARIT EDNA OHLENBUSCH, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

Building network-connected programs and distributed systems is a powerful way to provide availability in our digital, always-connected era. As always, however, with great power comes great complexity. As such, reasoning about distributed systems is well-known to be quite difficult.

In this paper we present Aneris, a state-of-the-art separation logic, capable of *node-local reasoning* about concurrent and distributed systems. The logic is higher-order, concurrent, with higher-order store and network sockets, built entirely in the Coq proof-assistant.

We use our logic to verify a load-balancer, that use threading to distribute load amongst servers, and to verify an implementation of the *two-phase-commit* protocol with a verified replicated logging service as client.

The two examples certifies that Aneris is well-suited for both horizontal and vertical modular programming and modular reasoning.

Additional Key Words and Phrases: distributed systems, separation logic, higher-order logic, concurrency, formal verification

1 INTRODUCTION

Network-connected applications, in particular distributed systems, are used every day by myriads of people for providing financial services, hailing taxis and sending messages over social media. However, formal reasoning about such systems is well-known to be difficult because of the complexity that concurrent, stateful programs expose. A well-known approach to combat complexity in programs is to disentangle software systems into independent modules to enable local reasoning. Local reasoning enables (a) information hiding and abstraction and (b) separation of concerns. For distributed systems this is known as a microservices architecture.

Previous work on verification of distributed systems has traditionally focused on verification of protocols of core network components. This approach has proven particularly useful within the context of model-checking, by validating both safety and liveness assertions [Pnueli 1977], such as SPIN Holzmann [1997], TLA+ [Lamport 1993] and Mace [Killian et al. 2007]. More recently, significant contributions has been made in the field of formal proofs of implementations of challenging protocols, such as two-phase-commit, lease-based key-value stores, Paxos and Raft [Hawblitzel et al. 2015; Lesani et al. 2016; Rahli et al. 2015; Sergey et al. 2017; Wilcox et al. 2015].

All of these developments define domain specific languages, specialized for distributed systems verification. Protocols and modules proven correct can be compiled to an executable often relying on some trusted code-base. However, reusing verified components in larger applications is difficult because: (1) the implementation specific code is often mixed with logical assertions about the abstract state, therefore lacking abstraction (a), (2) the DSL's do not support many modern language constructs such as concurrency or simplistic local state, (3) combining artifacts requires a unified

Authors' addresses: Morten Krogh-Jespersen, Aarhus University, Denmark, mkj@cs.au.dk; Amin Timany, imec-Distrinet, KU Leuven, Belgium, amin.timany@cs.kuleuven.be; Marit Edna Ohlenbusch, Aarhus University, Denmark, meo@cs.au.dk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

framework to reason about specifications written in separate DSL's. The Disel framework [Sergey et al. 2017] is a promising candidate for modular programming of distributed components, however, the language lacks features such as concurrency and node-local references are visible in protocol specifications.

In this work, we present Aneris, a framework for verifying real-world network-connected applications in Iris [Krebbers et al. 2017a], specifically developed with modular reasoning in mind. We obtain this property by what we refer to as **node-local reasoning**, similar to *thread-local reasoning* for thread-concurrent programs. We start by motivating the need for Aneris.

1.1 Why Aneris?

The design goal of Aneris, and the surface language, AnerisLang, is to facilitate verified modular programming of large software systems, including distributed systems.

(1) *The programming language should be realistic, familiar and easy to work with in practice*: In particular, AnerisLang should have higher-order functions, local state, concurrency and network primitives, to aid the developer in writing succinct, performant programs.
(2) *Separation of programming and proving*: Large developments are normally verified gradually. It should be allowed to used untrusted code and have verification be an option.
(3) *Code re-use and vertical composition of specifications*: By separating programming from proving, we already get modular programming on the implementation side. Additionally, a client should be able to use a verified component by only relying on the components specification, known as *modular reasoning* or vertical composition. This allows for changing and updating modules, without having to update the client or the clients proof.
(4) *Horizontal composition*: A verified component should be able to be composed horizontally, as long as the environment satisfies the protocols stated by the component. This allows for components to be composed horizontally and to build an verify large scale distributed systems.
(5) *Verification should be as easy as possible*: Interactive theorem proving, such as tactics in Coq, has proven successful for large scale verification projects. Ideally, using Aneris for verification purposes should provide a compatible experience.

Aneris is built on top of Iris, a state-of-the-art higher-order concurrent separation logic, which already has powerful built-in features to support reasoning about higher-order programs with higher-order store and concurrency, *e.g.*, higher-order impredicative invariants, higher-order Hoare triples, *etc.*. Setting up such a powerful program logic is difficult in general because it requires one to solve recursive domain equations – however, this has been solved in Iris. Thus, we can obtain all the features mentioned in bullet point (1) above by setting up Aneris on top of Iris. We discuss in Section 4.2 how Iris's facilities are used to define the program logic of Aneris.

There are many different ways of adding network primitives to languages. One approach is *message-passing*, either by first-class communication channels from the π-calculus or by an implementation of the actor model, similar to Erlang. However, any such implementation is an abstraction built on top of network sockets. Network sockets are a quintessential part of distributed systems and all major operating systems provide an application programming interface (API) for it. AnerisLang provides support for network sockets by exposing a simple API with the core methods necessary for UDP based programming. This allows for a wide-range of real-world systems and protocols to be written (and verified) in AnerisLang.

The higher-order concurrent distributed separation logic of Aneris provides a clear distinction between the programs, specifications for those programs and proofs thereof. This satisfies bullet

point (2) above. Hoare-style reasoning, a traditional way of giving abstract specifications to implementations, can be encoded in Iris and by extension Aneris through weakest-pre-conditions, allowing one to compose proofs about programs vertically without relying on specific components' implementation, satisfying (3).

Formal reasoning about nodes in distributed systems has often been done by giving an abstract model in the form of some kind of *state-transition system* or *flow-chart*, in the tradition of Floyd [1967]; Lamport [1977, 1978]. States are normally taken to be a view of the global state and events are then observable changes to this state. State-transition systems are quite versatile and have been used in other verification applications, *e.g.*, logical relations models [Ahmed et al. 2009; Dreyer et al. 2010] and in Hoare-style logic and type-theory [Sergey et al. 2017; Svendsen and Birkedal 2014]. However, Jung et al. [2015] showed that all you need is monoids (to encode resources) and invariants[1] (to encode protocols with the help of resources). We follow said approach, and associate each socket with a protocol in the form of an Aneris predicate on the incoming messages. We further allow the network to own the resources in transit, thus resources are transferred from the sender to the network before they are transferred to the recipient. This allows for independent program verification (state evolution) and local synchronization by ownership transfer. This enables reasoning about distributed systems in a fully node-local way (4).

Finally, by the virtue of using Iris as a basis for Aneris we have the Iris proof mode (IPM) [Krebbers et al. 2017b] at our disposal. It enables us to carry out interactive reasoning about the distributed separation logic of Aneris. This makes verification of distributed systems more pleasant and intuitive (5).

In summary, the key contributions of this work are:

- AnerisLang, a formalized higher-order functional programming language, with higher-order store, concurrency and network sockets, allowing for dynamic creation and binding of sockets to addresses with serialization and de-serialization primitives for encoding and parsing messages.
- Aneris, the first higher-order, concurrent, separation logic with support for network sockets, verified fully in the proof-assistant Coq. Since the logic is built on top of Iris, assertions on state and protocols can use all of the features from Iris, including invariants and monoids.
- A simple, novel, approach to guarding network sockets as predicates on messages, allowing for logical synchronization by the means of ownership-transfer. Ultimately, this enables what we refer to as **node-local reasoning**, the basic principle that allows for *modular reasoning* of distributed systems components.
- We use Aneris to verify a load-balancer, a program that distributes work on multiple servers by the means of threaded concurrency. The only assumption made on the servers is that they have a known address and that the socket protocols do not assert anything about the sender.
- We use Aneris to prove an implementation of *two-phase-commit* correct. Specifically, we prove that the coordinator and participant components satisfy the protocol. We then use these components in a distributed client of the two-phase-commit that does replicated logging, showing that vertical composition is achieved through node-local reasoning.

The structure of the rest of the paper. We start be describing the core concepts of Aneris in 2. We then show the operational semantics of AnerisLang 3 before showing adequacy and how to encode Aneris in Iris in 4. We then use the logic by showing a specification for a load-balancer 5 and two-phase-commit 6, before describing related work 7 and concluding 8.

[1]This has since been reduced to just resources as invariants can be, and are in recent version of Iris, implemented using high-order ghost resources [Jung et al. 2016; Krebbers et al. 2017a].

2 THE CORE CONCEPTS OF ANERIS

In this section, we present Aneris' approach to formal verification of distributed systems: node-local reasoning and protocols. These concepts are already familiar in the context of separation logic, thus we start by describing local reasoning in this context. We then explain how to lift thread-local reasoning to *node-local* reasoning, a novel approach to distributed systems verification. Finally, we describe protocols in Aneris and show a concrete lock server with a guarding protocol.

2.1 Local Reasoning and Thread Local Reasoning

Arguably the most important feature of (concurrent) separation logic, to which it owes its success and prevalence, is that it enables modular reasoning. Originally, separation logic [Reynolds 2002] was introduced to enable modular reasoning about the heap. The essential idea was that we could give a local specification $\{P\}\, e\, \{x.\ Q\}$ to a program e involving only the *footprint* of e. Local specifications could then be lifted to (more) global specifications by the following Frame-rule:

$$\textsc{Frame-rule}\quad \frac{\{P\}\, e\, \{x.\ Q\}}{\{P * R\}\, e\, \{x.\ Q * R\}}$$

Here, the proposition R is called *the frame*. The symbol $*$ is separating conjunction. Intuitively, $P * Q$ holds if resources (in this case heaps) can be divided into two disjoint resources such that P holds for one and Q holds for the other. Thus, the Frame-rule essentially says that executing e for which we know $\{P\}\, e\, \{x.\ Q\}$ cannot possibly affect parts of the heap that are *separate* from its footprint.

Ever since its introduction, separation logic has been extended to resources beyond the heap of the program. Concurrent separation logics [O'hearn 2007] have been developed to reason about concurrent programs and again a pre-eminent feature of these program logics is modular reasoning, in this instance with respect to concurrency, *i.e.*, *thread-local* reasoning. That is, when reasoning about a concurrent program we consider threads one at a time and need not reason about different interleavings explicitly. In a way, our frame here is, in addition to the shared fragments of heap and other resources, the execution of other threads which can be interleaved throughout the execution of the thread being verified. This can be seen in the following Fork-rule:

$$\textsc{Fork-rule}\quad \frac{\{P\}\, e\, \{x.\ \mathsf{true}\}}{\{P\}\, \mathsf{fork}\ \{e\}\ \{x.\ x = ()\}}$$

One notable program logic in the family of concurrent separation logics is Iris. Iris is a concurrent higher-order separation logic framework which was designed to reason about concurrent higher-order imperative programming languages. Iris has already proven quite versatile for reasoning about a number of sophisticated properties of programming languages, *e.g.*, [Jung et al. 2017; Kaiser et al. 2017; Timany et al. 2017]. In order to support modular reasoning about concurrent programs, Iris: (1) features *impredicative invariants* for expressing protocols among multiple threads, and, (2) allows encoding of *higher-order ghost state* using a form of partial commutative monoids for reasoning about resources. We will give examples of these features and explain them in more detail later on.

Aneris programs are higher-order imperative concurrent programs that run on multiple nodes in a distributed system. One of the main contributions of the present work is that when reasoning about distributed systems in Aneris, alongside *heap-local* and *thread-local* reasoning we reason *node-locally*. That is, when proving correctness of Aneris programs we reason about each node of the system in isolation.

In the rest of this section, we explain, at a conceptual level, how we achieve node-local reasoning in Aneris. The key idea is that although distributed systems and concurrent programs are vastly different, they conceptually share some essential features. This essential similarity allows us to put the machinery that Iris provides for thread-reasoning about concurrent programs into use for node-local reasoning about distributed systems.

2.2 Node-Local Reasoning About Distributed Systems

By the virtue of working in Iris, Aneris reasoning is both modular with respect to separation logic frames and with respect to threads. Similarly to threads Aneris allows for *node-local* reasoning about programs:

$$\textsc{Start-rule}\quad \frac{\{P * \mathit{freePorts}(\mathit{ip}, \{p | 0 \le p \le 65536\})\}\ \langle n; e\rangle\ \{x.\ \mathsf{true}\}}{\{P * \mathit{freeIp}(\mathit{ip})\}\ \langle \mathfrak{S}; \mathsf{start}\ \{n; \mathit{ip}; e\}\rangle\ \{x.\ x = \langle \mathfrak{S}; ()\rangle\}}$$

Here start is the command that launches a new node named n in the distributed system associated with ip-address ip running program e. Note that only the distinguished system node $\mathfrak{S}$ can start new nodes. The idea is that in Aneris, the execution of the system starts with the execution of $\mathfrak{S}$ as the only node in the distributed system. $\mathfrak{S}$ then bootstraps the distributed system by starting other nodes. In order to start a new node associated with ip-address ip, one needs to provide $\mathit{freeIp}(\mathit{ip})$ which indicates that ip-address ip is not used by other nodes. The node can on the other hand rely on the fact that when it starts, all ports on the ip-address ip are available. Notice that to facilitate modular reasoning, free ports can be divided up:

$$\frac{A \cap B = \emptyset}{\mathit{freePorts}(\mathit{ip}, A) * \mathit{freePorts}(\mathit{ip}, B) \dashv\vdash \mathit{freePorts}(\mathit{ip}, A \cup B)}$$

where $\dashv\vdash$ is logical equivalence of Iris propositions.

In Aneris we associate with each socket (pair of ip-addresses and ports) a protocol which restricts what can be communicated over that socket. In Aneris terms, we write $s \Rightarrow^{\mathsf{prot}} \Phi$ to mean that socket s is governed by the protocol Φ. In particular, if we have $s \Rightarrow^{\mathsf{prot}} \Phi$ and $s \Rightarrow^{\mathsf{prot}} \Psi$, we can conclude that Φ and Ψ are the same protocol.

We support two kinds of sockets: static sockets and dynamic sockets. This distinction is an abstract, it is only at the level of the logic and not the distributed system itself. Static sockets are those which have primordial protocols agreed upon before starting the system. The static protocols are primarily for addresses pointing to servers. By having a primordial protocol, any node in the system (including the server itself) know and must respect this protocol.

To support node modular reasoning in general we distinguish static and dynamic addresses. To this end, we use the proposition $\overset{f}{\mapsto}(A)$ which means the set of addresses in A are static and should have a fixed interpretation. The proposition $\overset{f}{\mapsto}(A)$ expresses knowledge without asserting ownership of resources. In Iris terminology, this proposition is *persistent*: $\overset{f}{\mapsto}(A) \dashv\vdash \overset{f}{\mapsto}(A) * \overset{f}{\mapsto}(A)$.

Corresponding to the two kinds of addresses, we have the two rules Bind-stat-rule and Bind-dyn-rule shown below for binding addresses (starting the communication over) to a socket. Notice that in the case of Bind-dyn-rule one can choose the protocol while in the case of Bind-stat-rule the protocol is existentially quantified, *i.e.*, it is the primordial protocol associated to (ip, p).

BIND-STAT-RULE

$$\left\{ \overset{f}{\mapsto} (A) * (ip, p) \in A * z \overset{s\ [n]}{\mapsto} \text{None} * \mathit{freePorts}(ip, \{p\}) \right\}$$
$$\langle n; \text{socketbind}\, z\, (ip, p) \rangle$$
$$\left\{ x.\ x = 0 * \exists \Phi.\ z_n \overset{s\ [n]}{\mapsto} \text{Some}\,(ip, p) * (ip, p) \Rrightarrow^{\text{prot}} \Phi \right\}$$

BIND-DYN-RULE

$$\left\{ \overset{f}{\mapsto} (A) * (ip, p) \notin A * z_n \overset{s\ [n]}{\mapsto} \text{None} * \mathit{freePorts}(ip, \{p\}) \right\}$$
$$\langle n; \text{socketbind}\, z\, (ip, p) \rangle$$
$$\left\{ x.\ x = 0 * z \overset{s\ [n]}{\mapsto} \text{Some}\,(ip, p) * (ip, p) \Rrightarrow^{\text{prot}} \Phi \right\}$$

Here, $z \overset{s\ [n]}{\mapsto} (ip, p)$ is a socket assertion used to keep track of resources associated with the socket that belongs to node n. The resource is there to ensure that each socket is bound only once.

2.3 A Lock Server

Mutual exclusion in distributed systems is often a necessity and there are many different approaches for providing it. The simplest solution, presented in this section, is a centralized algorithm with a single node acting as a coordinator. The code is shown in Figure 1. We later show a more involved example of log-replication using two-phase-commit (6).

```
rec lockserver ip p :=
  let lock := ref NONE in
  let skt := socket in
  Socketbind skt (makeaddress ip p);
  listen skt (rec h msg from :=
    if msg = "LOCK"
    then match !lock with
         NONE => owner ← SOME ();
                 sendto skt "YES" from
       | SOME _ => sendto skt "NO" from
         end
    else owner ← NONE;
         sendto skt "RELEASED" from);
    listen skt h)
```

```
rec listen skt handler :=
  match receivefrom skt with
    SOME m => handler (fst m)
                      (snd m) in
  | NONE => listen skt handler
  end
```

Fig. 1. A lock server in AnerisLang. Function binders are strings in Coq but are shown as regular binders for the sake of clarity.

The lock server declares a node-local variable `lock` to keep track of the lock. It then binds a new socket `skt` on the given address ip:port and continuously listens for incoming messages on the socket. When a "LOCK" message arrives and the lock is available, the lock is taken and the server responds "YES". If the lock was already taken the server responds with "NO". Finally, if the request is not "LOCK", the lock is released and the server responds with "RELEASED".

Notice that the lock server program looks as if it was written in a decent functional language with sockets. Messages sent and received are strings to make programming with sockets easier (similar to send_substring in the Unix module in OCaml). This is a direct result of the ambition to make AnerisLang useful for verifying existing code but also writing new programs in AnerisLang.

A node-local specification for the lock server, with a universally quantified lock source R and a protocol governing the socket endpoint $(ip, p) \Mapsto^{\mathsf{prot}} \phi$, is as follows:

$$\left\{R * (ip, p) \Mapsto^{\mathsf{prot}} \phi * \overset{f}{\mapsto} (\{(ip, p)\} \cup A) * \mathit{freePorts}(ip, \{p\})\right\}$$
$$\langle n; \mathsf{lockserver}\,()\rangle$$
$$\{\mathsf{True}\}$$

There are several interesting observations one can make on the lock server example:

- The lock server can allocate, read and write node-local references but these are hidden in the specification.
- Sockets can be created and bound to specified endpoints. In this example we expect the lock server to be primordial, *i.e.*, the system should agree on a protocol $(ip, p) \Mapsto^{\mathsf{prot}} \phi$. Notice as well that there are no channel descriptors or assertions on the socket in the code.
- Without a proper protocol, the lock server fails to provide mutual exclusion since everyone can release the lock.

With the necessity of protocols on sockets established, we explain Aneris' interpretation of sockets and protocols in more detail.

2.4 Aneris Sockets and Protocols

Conceptually, a socket is an abstract representation of a handle for a local endpoint of some channel. In Aneris we further restrict channels to use the User Datagram Protocol (UDP), which is *asynchronous*, *connectionless* and *stateless*. In accordance with UDP, Aneris provides no guarantee of delivery or ordering, although we assume duplicate protection, since spatial resources could otherwise potentially be duplicated. One can therefore think of Aneris sockets as open-ended multi-party communication channels without synchronization.

It is noteworthy that inter-process communication can happen in multiple ways in Aneris. Thread-concurrent programs can communicate through the store but they can also communicate by sending messages through sockets. For memory separated programs, there is no shared state and all communication is done with message-passing through sockets.

As mentioned in the Introduction, Iris has an un-opinionated approach to protocol design, therefore one can use a state-transition system if desired. However, modeling the execution of concurrent code with asynchronous message passing as state transition systems on thread and node levels can be quite cumbersome as opposed to a more descriptive approach in the form of resource reasoning.

A simple, novel approach to protocol design is to just give an Aneris predicate over messages to be received on the sockets. This can be regarded as a socket-local approach by requiring each socket to be guarded by an Aneris predicate. One can think of this as rely-reasoning, restricting the distributed environment's interference with a node on a particular socket. Ultimately, this is what allows node-local verification of programs.

Concretely, the socket protocol for a lock server can be specified as follows:

$$\mathit{lock}(m, \phi) \triangleq \mathit{body}(m) = \text{"}\mathsf{LOCK}\text{"} * ((\forall m'.\, \mathit{body}(m') = \text{"}\mathsf{NO}\text{"} \vee \mathit{body}(m') = \text{"}\mathsf{YES}\text{"} * R) \mathrel{-\!\!*} \phi(m'))$$
$$\mathit{rel}(m, \phi) \triangleq \mathit{body}(m) = \text{"}\mathsf{RELEASE}\text{"} * R * (\forall m'.\, \mathit{body}(m') = \text{"}\mathsf{RELEASED}\text{"} \mathrel{-\!\!*} \phi(m))$$
$$\mathit{lock\_si} \triangleq \lambda m.\, \exists \phi.\, a \Mapsto^{\mathsf{prot}} \phi * (\mathit{lock}(m, \phi) \vee \mathit{rel}(m, \phi))$$

The resources describing the lock, R, are transferred to the client if the server responds "YES" and the same resources must be returned when calling "RELEASE". The protocol is sufficient to prove that the clients of distributed system will use the lock server correctly.

$$
\begin{aligned}
v &::= () \mid \mathsf{true} \mid \mathsf{false} \mid i \mid s \mid \ell \mid z \mid \mathsf{rec}\, f(x) = e \mid (v, v) \mid \mathsf{inj}_i\, v \mid \mathsf{address}\; s\; p \\
e &::= v \mid \boxdot\, e \mid e \circledcirc e \mid e\; e \mid \mathsf{find}\; e\; e\; e \mid \mathsf{substring}\; e\; e\; e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid (e, e) \mid \pi_i\, e \mid \mathsf{inj}_i\, e \\
&\quad \mid \mathsf{match}\, e\, \mathsf{with}\, \mathsf{inj}_i\, x \Rightarrow e_i\, \mathsf{end} \mid \mathsf{ref}(e) \mid\, !\, e \mid e \leftarrow e \mid \mathsf{cas}(e, e, e) \mid \mathsf{fork}\; \{e\} \\
&\quad \mid \mathsf{start}\; \{n; e; e\} \mid \mathsf{makeaddress}\; e\; e \mid \mathsf{socket} \mid \mathsf{socketbind}\; e\; e \mid \mathsf{sendto}\; e\; e\; e \mid \mathsf{receivefrom}\; e
\end{aligned}
$$

Fig. 2. Syntax of AnerisLang

Additionally, the lock protocol also illustrates how primordial servers respond to dynamic bound sockets. As mentioned earlier, the lock server socket should be primordial, however, the lock does not need to know about its clients as long as the clients follow the socket protocol defined by the lock server. As a consequence, a client has to prove that she can receive a reply from the server. This is specifically done by proving the resource-aware implication $-\!\!*$ known as magic wand (expanded upon in 4).

The lock server protocol is very similar to a non-blocking specification of a locking module in concurrent separation logic, where the implication can be seen as the post-condition.

3 OPERATIONAL SEMANTICS OF ANERISLANG

In this section, we present AnerisLang, a feature-rich concurrent programming language with network primitives. Usually, in a concurrent programming language, the operational semantics of the program are defined as a reduction relation over configurations consisting of pairs of a state, *i.e.*, heap, and a thread pool. AnerisLang is a programming language designed to model, reason about and implement distributed systems. These are systems comprised of a number of nodes, each of which concurrently runs a number of threads. Hence, configurations of AnerisLang are pairs consisting of state, *i.e.*, a heap for each node in the system and the state of the network, together with a collection of thread pools, one for each node.

In the sequel, we first present the syntax of AnerisLang. Subsequently, we show a *head-step* relation for programs. This head-step is then lifted to arbitrary AnerisLang configurations in the usual way in three steps: (1) a program takes a step if one of its nodes takes the corresponding step, (2) a node takes a step if one of its concurrently running threads does, and (3) a thread makes a step if the sub-expression of the program running on that thread in the evaluation position takes a head-step.

3.1 Syntax of AnerisLang

AnerisLang is a call-by-value, higher-order, concurrent imperative programming language with higher-order mutable references, fine-grained concurrency and network sockets. The syntax for values and expressions is shown in Figure 2.

In this figure v ranges over values and e ranges over expressions. In addition to the standard literal values we write i for integers and s for strings. The value $\mathsf{address}\; s\; p$ is a network address where s is the ip-address and p is the port number. Booleans, integers and strings can be manipulated by unary operations $\boxdot$ (negation, unary minus, string-length and conversions between strings and integers) and binary operations $\circledcirc$ (arithmetic operations, comparisons and test for equality). The string operations find and substring find the index of a particular substring and split a string producing a substring respectively.

We use ℓ for memory locations. These can be allocated by ref, read by $!\, l$, and updated by $l \leftarrow v$. The expression $\mathsf{fork}\; \{e\}$ forks off a new thread on the node it is running on. The atomic *compare-and-set* operation, $\mathsf{cas}(l, v_1, v_2)$, is used to achieve synchronization between threads on a specific

$$
\begin{aligned}
n \in \mathit{Node}, \mathit{Ip} &\triangleq \mathit{String} \\
a \in \mathit{Address} &\triangleq \mathit{Ip} \times \mathit{Port} \\
\mathit{MessageState} &\triangleq \{\textsc{Sent}, \textsc{Received}\} \\
\mathit{Message} &\triangleq \mathit{Address} \times \mathit{Address} \times \mathit{String} \times \mathit{MessageState} \\
\mathit{MessageStable} &\triangleq \mathit{Address} \times \mathit{Address} \times \mathit{String} \\
h \in \mathit{Heap} &\triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val} \\
S \in \mathit{Sockets} &\triangleq \mathit{Handle} \xrightarrow{\text{fin}} \mathit{Address}^? \\
\mathit{Lookup} &\triangleq \mathit{Address} \xrightarrow{\text{fin}} \mathit{Node} \\
\mathit{Ports} &\triangleq \mathit{Ip} \xrightarrow{\text{fin}} \wp^{\text{fin}}(\mathit{Port}) \\
\mathit{MessageSoup} &\triangleq \mathit{MessageId} \xrightarrow{\text{fin}} \mathit{Message} \\
\mathit{NetworkState} &\triangleq \mathit{Node} \xrightarrow{\text{fin}} \mathit{Heap} \times \mathit{Node} \xrightarrow{\text{fin}} \mathit{Sockets} \times \mathit{Lookup} \times \mathit{Ports} \times \mathit{MessageSoup}
\end{aligned}
$$

Fig. 3. The physical state interpretation of Aneris

memory location l. Operationally, it does the following *atomically*: it tests whether ℓ has value v_1 and if so, updates the location to v_2. It returns true if successful and false otherwise.

The expression start $\{n; ip; e\}$ starts a new node n running program e assigned the ip-address ip. The only stipulation here is that nodes can only be started at the bootstrapping phase of the distributed system by a special system-node $\mathfrak{S}$.

We use z to range over socket handles. The network primitives socket, socketbind, sendto and receivefrom correspond to the BSD-sockets API methods socket, to create sockets, bind, to bind to a socket, sendto, to send over the network on a socket, and, recvfrom, to receive over the network on socket, respectively. The expression makeaddress is used to compute addresses. The expression makeaddress $e\ e'$ evaluates to a network address if e evaluates to a string for the ip address and e' evaluates to a number for the port number [2].

3.2 Semantics of AnerisLang

We define the operational semantics of AnerisLang in three stages. First we define a node-local thread-local, head-step relation between configurations $(e, h, S) \rightarrow_{n,h} (e', h', S')$ for a heap h, allocated socket-endpoints S and expressions e, a node n and evaluation context $K \in \mathit{Ectx}$. Heaps are finite maps from locations to values and sockets are finite maps from socket handles to option socket address. We write $K[e]$ to denote the expression of plugging e into the K and $e[v/x]$ to denote capture-avoiding substitution of v for x in e. Evaluation contexts are listed in Figure 4 and an excerpt of the node-local rules is shown in Figure 5.

Node-local steps are lifted to a network-aware stepping relation $\rightarrow_h$ by lifting expressions to node-expressions $\epsilon = \langle n; e \rangle$ and lifting the local state $\mathit{Heap} \times \mathit{Sockets}$ to $\mathit{NetworkState}$, tracking heaps H and sockets Z for all nodes, all bound addresses in the system L, ports in use P and messages sent M. The network-aware stepping relation is shown in Figure 6.

[2] In the setting of OCaml, makeaddress would internally call `inet_addr_of_string`

$$
\begin{aligned}
K ::= {} & [] \mid \boxdot\ K \mid K \circledcirc e \mid v \circledcirc K \mid K\ e \mid v\ K \mid \mathsf{find}\ K\ e\ e \mid \mathsf{find}\ v\ K\ e \mid \mathsf{find}\ v\ v\ K \mid \mathsf{substring}\ K\ e\ e \\
& \mid \mathsf{substring}\ v\ K\ e \mid \mathsf{substring}\ v\ v\ K \mid \mathsf{if}\ K\ \mathsf{then}\ e\ \mathsf{else}\ e \mid (K, e) \mid (v, K) \mid \pi_i\ K \mid \mathsf{inj}_i\ K \\
& \mid \mathsf{match}\ K\ \mathsf{with}\ \mathsf{inj}_i\ x \Rightarrow e_i\ \mathsf{end} \mid \mathsf{ref}(K) \mid\ !\,K \mid K \leftarrow e \mid v \leftarrow K \mid \mathsf{cas}(K, e, e) \mid \mathsf{cas}(v, K, e) \\
& \mid \mathsf{cas}(v, v, K) \mid \mathsf{makeaddress}\ K\ e \mid \mathsf{makeaddress}\ v\ K \mid \mathsf{socketbind}\ K\ e \mid \mathsf{socketbind}\ v\ K \\
& \mid \mathsf{sendto}\ K\ e\ e \mid \mathsf{sendto}\ v\ K\ e \mid \mathsf{sendto}\ v\ v\ K \mid \mathsf{receivefrom}\ K
\end{aligned}
$$

Fig. 4. Evaluation contexts of AnerisLang

$$((\mathsf{rec}\ f(x) = e)\ v, h, S) \to_{n,h} (e[v, (\mathsf{rec}\ f(x) = e)/x, f], h, S)$$

$$(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3, h, S) \to_{n,h} (e_2, h, S) \qquad (\pi_1\ (v_1, v_2), h, S) \to_{n,h} (v_1, h, S)$$

$$\frac{\ell \notin \mathrm{dom}(h)}{(\mathsf{ref}(v), h, S) \to_{n,h} (\ell, h \uplus \{\ell \mapsto v\}, S)} \qquad \frac{h = h' \uplus \{\ell \mapsto v'\}}{(\ell \leftarrow v, h, S) \to_{n,h} ((), h' \uplus \{\ell \mapsto v\}, S)}$$

$$\frac{v = h(\ell)}{(!\,\ell, h, S) \to_{n,h} (v, h, S)} \qquad \frac{h = h' \uplus \{\ell \mapsto v\}}{(\mathsf{cas}(\ell, v, v'), h, S) \to_{n,h} (\mathsf{true}, h' \uplus \{\ell \mapsto v'\})}$$

$$\frac{h = h' \uplus \{\ell \mapsto v''\} \qquad v \neq v''}{(\mathsf{cas}(\ell, v, v'), h, S) \to_{n,h} (\mathsf{false}, h, S)} \qquad \frac{h = h' \uplus \{\ell \mapsto v''\} \qquad v \neq v''}{(\mathsf{cas}(\ell, v, v'), h, S) \to_{n,h} (\mathsf{false}, h, S)}$$

$$\frac{z \notin \mathrm{dom}(S)}{(\mathsf{socket}, h, S) \to_{n,h} (z, h, S \uplus \{z \mapsto \mathsf{None}\})}$$

Fig. 5. An excerpt of the node-local head-reduction rules

The rule for socketbind expresses that a socket can be bound if it is not already bound, the address is not already used in the network and the port is not in use at the address. Hereafter, the address is stored in the lookup table L and the port is no longer available in P. One can think of L as a global router making sure address-conflicts do not occur.

The rule for sending messages through a bound socket is sendto (there is also a version for unbound sending, see the Coq development for the gory details). The message is populated with the sender's address (*from*), a destination address (*to*), the message itself and a status flag indicating sent (and not received). The operation returns the number of characters sent. We use $-$ to indicate no other changes. Notice that we do not care if the destination exists or not.

Finally, messages can be received by the receivefrom rule. A socket can receive messages if it has a registered address and awaiting messages. Upon receiving the message, the message and sender are returned and the status flag of the message is updated to Received by the *rec* function. If no messages are waiting, None is returned.

The final head-step relation is a *distributed systems* relation $\rightarrow$ shown below. The distributed systems relation reduces by picking a thread on any node or forking off a new thread.

$$\frac{(e, h, S) \rightarrow_{n,h} (e', h', S')}{\langle n; e\rangle, (H[n \mapsto h], Z[n \mapsto S], L, P, M) \rightarrow_h \langle n; e'\rangle, (H[n \mapsto h'], Z[n \mapsto S'], L, P, M)}$$

$$\frac{Z(n)(z) = \text{None} \qquad (ip, p) \notin \text{dom}(L) \qquad p \notin P(ip) \qquad Z' = Z[n \mapsto S[z \mapsto \text{Some } a]]}{\langle n; \text{socketbind } z\ a\rangle, (H, Z, L, P, M) \rightarrow_h \langle n; 0\rangle, (H, Z', L[a \mapsto n], P[a \mapsto P(a) \cup \{p\}], M)}$$

$$\frac{Z(n)(z) = \text{Some } from \qquad m = (from, to, msg, \textsc{Sent}) \qquad m_{id} \notin M}{\langle n; \text{sendto } z\ msg\ to\rangle, (-, M) \rightarrow_h \langle n; \text{length } msg\rangle, (-, M[m_{id} \mapsto m])}$$

$$\frac{Z(n)(z) = \text{Some } a \qquad m_{id} \mapsto m \in \{m_{id} \mapsto m \mid m \in M \wedge to(m) = a \wedge state(m) = \textsc{Sent}\}}{\langle n; \text{receivefrom } z\rangle, (-, M) \rightarrow_h \langle n; \text{Some } (msg(m), from(m))\rangle, (-, M[m_{id} \mapsto rec(m)])}$$

$$\frac{Z(n)(z) = \text{Some } a \qquad \emptyset = \{m_{id} \mapsto m \mid m \in M \wedge to(m) = a \wedge state(m) = \textsc{Sent}\}}{\langle n; \text{receivefrom } z\rangle, (-, M) \rightarrow_h \langle n; \text{None}\rangle, (-, M[m_{id} \mapsto rec(m)])}$$

Fig. 6. Network-aware head-reduction rules for bound sockets.

$$\frac{(\langle n; e\rangle, \sigma) \rightarrow_h (\langle n; e'\rangle, \sigma')}{(\vec{\epsilon}_1, \langle n; K[e]\rangle, \vec{\epsilon}_2; \sigma) \rightarrow (\vec{\epsilon}_1, \langle n; K[e']\rangle, \vec{\epsilon}_2; \sigma')}$$

$$(\vec{\epsilon}_1, \langle n; K[\text{fork } \{e\}\ ]\rangle, \vec{\epsilon}_2; \sigma) \rightarrow (\vec{\epsilon}_1, \langle n; K[()]\rangle, \vec{\epsilon}_2, \langle n; e\rangle; \sigma)$$

4 SEMANTICS OF ANERIS

With the Semantics of AnerisLang defined we focus on building the logic for Aneris in this section. We start by introducing the Iris-logic, before presenting the Aneris logic and stating adequacy.

4.1 A Primer on Iris

Iris was originally presented as a framework for higher-order (concurrent) separation logic inside the proof assistant Coq. It has a built-in notion of physical state, ghost-state (monoids) and invariants. Moreover, it includes weakest preconditions which are useful for Hoare-style reasoning concurrent imperative programs [Jung et al. 2015]. Recently, a simpler Iris *base logic* was defined. This base logic suffices for defining all earlier built-in concepts, such as invariants, higher-order ghost state and weakest preconditions. [Krebbers et al. 2017a]. It is this latter version of Iris we depend on for defining Aneris.

The quantifiable types of Iris, denoted by κ, are shown below:

$$\kappa ::= 1 \mid \kappa \times \kappa \mid \kappa \rightarrow \kappa \mid \mathbb{N} \mid \mathbb{B} \mid \mathit{Ectx} \mid \mathit{Var} \mid \mathit{Expr} \mid \mathit{Val} \mid \kappa \xrightarrow{\text{fin}} \kappa \mid \text{finset}(\kappa) \mid \mathit{Monoid} \mid \mathit{Names} \mid \mathit{iProp} \mid \ldots$$

Iris includes basic types such as the unit, 1, pairs and ordinary functions. $\mathbb{N}$ is the type of natural numbers, $\mathbb{B}$ is the type of booleans and *Expr* and *Val* are the types of AnerisLang syntactic expressions and values. The type $\kappa \xrightarrow{\text{fin}} \kappa$ is that of partial functions with finite support and finset(κ) is the type of finite sets. *Monoid* is the type of monoids, *Names* is the type of ghost names,

and *iProp* is the type of Iris propositions. An excerpt of the grammar for Iris propositions P is:

$$P ::= \top \mid \bot \mid P * P \mid P \mathrel{-\!\!*} P \mid P \wedge P \mid P \Rightarrow P \mid P \vee P \mid \forall x : \kappa.\ \Phi \mid \exists x : \kappa.\ \Phi$$
$$\mid \Box P \mid \mathsf{wp}\ \langle n; e\rangle \left\{x.\ P\right\} \mid \{P\}\ \langle n; e\rangle\ \{x.\ Q\} \mid \Rrightarrow P \mid \boxed{P}^{\mathcal{N}} \mid \ldots$$

The grammar includes the usual connectives of higher-order separation logic ($\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$, $*$, $\mathrel{-\!\!*}$, $\forall$ and $\exists$). In this grammar Φ is an Iris predicate, *i.e.*, a term of type $\kappa \to \mathit{iProp}$ (for an appropriate κ). Intuitively, as in any separation logic, propositions P denote sets of resources satisfied by P. Propositions joined by separating conjunction, $P * P'$, express that resources can be split into disjoint parts; one satisfying P and the other satisfying P'. Propositions $P \mathrel{-\!\!*} P'$ describe those resources that, combined with disjoint resources satisfied by P, can result in resources satisfied by P'. In addition to these standard separation logic connectives, Iris includes other useful connectives of which the most frequently used ones will be described below.

Arguably, the most important feature of any separation logic is the ability to update the parts you own while leaving all other resources intact. In Iris this is accomplished by the "update" modality[3], $\Rrightarrow$. Intuitively, $\Rrightarrow P$ holds for those resources that can be updated to resources that satisfy P, without violating the environment's knowledge or ownership of resources. The update modality is idempotent, $\Rrightarrow(\Rrightarrow P) \dashv\vdash \Rrightarrow P$. We write $P \Mrightarrow Q$ as a shorthand for $P \mathrel{-\!\!*} \Rrightarrow Q$.

Possibly, the second most important feature of any separation logic is the concept of invariants. Conceptually, invariants capture what is known, as opposed to what is owned. In Iris, the "persistence" modality ($\Box$) is used to capture a sublogic of knowledge that obeys standard rules for intuitionistic, higher-order logic. We say that a proposition is *persistent* if $P \vdash \Box P$. Intuitively, $\Box P$ are those propositions that are satisfied without asserting any exclusive ownership. Consequently, because $\Box P$ asserts no exclusive ownership, it is a duplicable assertion $(\Box P) * (\Box P) \dashv\vdash \Box P$. The persistence modality is idempotent. That is, $\Box P \vdash \Box\Box P$ for any P. Furthermore, $\Box P \vdash P$. The persistence modality also commutes with all of the connectives of higher-order separation logic.

The $\triangleright$ modality, pronounced "later", is an abstraction of step-indexing [Appel and McAllester 2001; Appel et al. 2007; Dreyer et al. 2011]. For any proposition P, we have that $P \vdash \triangleright P$, which in terms of step-indexing means that if P holds now then it also does so a step later. In Iris, the $\triangleright$ modality is used in the definition of weakest preconditions and to guard impredicative invariants to avoid self-referential paradoxes [Krebbers et al. 2017a]. The later modality commutes with all of the connectives of higher-order separation logic.

The propositions $\mathsf{wp}\ \langle n; e\rangle \left\{x.\ P\right\}$ and $\{P\}\ \langle n; e\rangle\ \{x.\ Q\}$ are Iris's propositions for reasoning about programs. Intuitively, a weakest precondition $\mathsf{wp}\ \langle n; e\rangle \left\{x.\ P\right\}$ holds when the expression e is *safe* to execute, *i.e.*, it does not get stuck, and that whenever it reduces to a value v, then $P[v/x]$ holds. In Iris, Hoare triples are defined based on weakest preconditions in the usual manner:

$$\{P\}\ \langle n; e\rangle\ \{x.\ Q\} \triangleq \Box\left(P \mathrel{-\!\!*} \mathsf{wp}\ \langle n; e\rangle \left\{x.\ Q\right\}\right)$$

Notice that we require Hoare triples to be persistent, *i.e.*, Hoare triples assert only knowledge and no ownership of resources. In other words, the Hoare triple $\{P\}\ \langle n; e\rangle\ \{x.\ Q\}$ states that all the (non-persistent) resources that are used by e are contained in P. At any point in the course of proving correctness of a program we can update resources. This is captured in the following property of the weakest preconditions:

$$\Rrightarrow \mathsf{wp}\ \langle n; e\rangle \left\{x.\ P\right\} \dashv\vdash \mathsf{wp}\ \langle n; e\rangle \left\{x.\ P\right\} \dashv\vdash \mathsf{wp}\ \langle n; e\rangle \left\{x.\ \Rrightarrow P\right\}$$

[3] In [Krebbers et al. 2017a] this modality is called the *fancy* update modality. Technically, this modality comes equipped with certain "masks" but we do not discuss those here.

The weakest precondition for values is equivalent to the postcondition holding for that value, modulo updating resources:

$$\textsc{wp-value}\quad \mathsf{wp}\,\langle n; v\rangle\,\{x.\ P\} \dashv\vdash \Rrightarrow P[v/x]$$

Proving safety of a program under an evaluation context, $K[e]$, can be done in two separate steps: (1) we prove that e is safe, and, (2) for a value w that we get out of the execution of e, $K[w]$ is safe to execute. This fact is embodied in Iris as the 4.1 rule below:

$$\textsc{wp-bind}\quad \frac{\mathsf{wp}\,\langle n; e\rangle\,\{y.\ \mathsf{wp}\,\langle n; K[y]\rangle\,\{x.\ P\}\}}{\mathsf{wp}\,\langle n; K[e]\rangle\,\{x.\ P\}}$$

Iris features invariants $\boxed{P}^{\mathcal{N}}$, pronounced invariantly P. The name $\mathcal{N}$ is the name associated with the invariant and is used to keep track of which invariants are opened as opening invariants multiple times in a nested fashion is in general unsound.[4] Invariants are Iris's way of encoding protocols for shared resources. Invariants, once established, can only be violated during the execution of an atomic program step, *i.e.*, the time when it cannot be noticed by other threads. The following two rules allow us to establish and open invariants:

$$\textsc{inv-alloc}\quad \frac{\triangleright P}{\Rrightarrow \boxed{P}^{\mathcal{N}}} \qquad \textsc{inv-open}\quad \frac{\boxed{P}^{\mathcal{N}} \qquad \triangleright P \mathrel{-\!\!*} \mathsf{wp}\,\langle n; e\rangle\,\{x.\ \triangleright P * Q\} \qquad e \text{ is physically atomic}}{\mathsf{wp}\,\langle n; e\rangle\,\{x.\ Q\}}$$

Note the use of the later modality with invariants. This is necessary as invariants are impredicative, *i.e.*, $\boxed{P}^{\mathcal{N}}$ is an *iProp* and so is P. The later modality is necessary [Krebbers et al. 2017a] to ensure that invariants do not allow encoding of self-referential paradoxes. Invariants are persistent as they simply assert the knowledge that some proposition holds invariantly.

4.2 Aneris: the program logic

Iris as a (program) logic consists of two layers: the Iris base logic, explained above in Subsection 4.1 except for weakest preconditions and Hoare triples, and a program logic defined on top of the base logic. The program logic layer of Iris is language agnostic. That is, the user of Iris can specify the syntax and operational semantics of their programming language together with the resources one needs to keep track of the state of programs (the heaps of individual nodes and the state of the network in our case) and as a result get a program logic, *i.e.*, a basic notion of weakest preconditions for that language. This is what we have done for Aneris: we have instantiated Iris's program logic layer with the syntax and operational semantics of Aneris given in Section 3.1.

The basic notion of weakest preconditions provided by the program logic layer of Iris is defined using the Iris base logic, based on the given operational semantics. In order to get a full-blown program logic one needs to derive the intended rules of the logic based on the definition of weakest preconditions provided by Iris.[5] In the sequel we present the weakest precondition rules that we derive for Aneris. We use these rules for proving correctness of the programs that we discuss in this paper.

[4] Indeed the update modality is annotated with a mask consisting of the set of names of invariants that are opened before and after the update to prevent such unsound opening of invariants. We omit masks of the update modality in this paper for the sake of brevity and simplicity.

[5] Iris as a program logic framework comes with an internal programming language. For this programming language, the Iris program logic is instantiated and all the relevant program logic rules are derived. Here, we put this internal language aside and start from scratch: we define syntax and semantics of Aneris and derive all the program logic proof rules presented in this section.

Aneris program logic rules can be divided into three classes: those pertaining to pure computations, those for manipulating the heap (of a node) and those for network-communications. We discuss them in that order.

The weakest precondition rules for pure computations are exactly as one would expect. An instructive excerpt is given below:

$$\textsc{fst-wp}\ \frac{\triangleright\, \mathsf{wp}\ \langle n; v\rangle\ \{\Phi\}}{\mathsf{wp}\ \langle n; \pi_1\,(v, w)\rangle\ \{\Phi\}} \qquad \textsc{if-true-wp}\ \frac{\triangleright\, \mathsf{wp}\ \langle n; e\rangle\ \{\Phi\}}{\mathsf{wp}\ \langle n; \mathsf{if\ true\ then}\ e\ \mathsf{else}\ e'\rangle\ \{\Phi\}}$$

$$\textsc{rec-wp}\ \frac{\triangleright\, \mathsf{wp}\ \langle n; e[\mathsf{rec}\ f(x) = e, v/f, x]\rangle\ \{\Phi\}}{\mathsf{wp}\ \langle n; (\mathsf{rec}\ f(x) = e)\ v\rangle\ \{\Phi\}}$$

Note that all these programs take a step of computation to execute and hence their antecedent is only required to hold at a later step.

The weakest preconditions for heap-manipulating programs are similar to their analogues in standard separation logic for ML-like programming languages:

$$\textsc{wp-alloc}\ \frac{\forall \ell.\ \ell \mapsto^{[n]} v \mathrel{-\!\!*} \mathsf{wp}\ \langle n; \ell\rangle\ \{\Phi\} \qquad \triangleright\, \mathit{IsNode}(n)}{\mathsf{wp}\ \langle n; \mathsf{ref}(v)\rangle\ \{\Phi\}} \qquad \textsc{wp-load}\ \frac{\ell \mapsto^{[n]}_q v \mathrel{-\!\!*} \mathsf{wp}\ \langle n; v\rangle\ \{\Phi\} \qquad \triangleright\, \ell \mapsto^{[n]}_q v}{\mathsf{wp}\ \langle n; !\,\ell\rangle\ \{\Phi\}}$$

$$\textsc{wp-store}\ \frac{\ell \mapsto^{[n]} w \mathrel{-\!\!*} \mathsf{wp}\ \langle n; ()\rangle\ \{\Phi\} \qquad \triangleright\, \ell \mapsto^{[n]} v}{\mathsf{wp}\ \langle n; \ell \leftarrow w\rangle\ \{\Phi\}} \qquad \textsc{wp-cas-suc}\ \frac{\ell \mapsto^{[n]} w \mathrel{-\!\!*} \mathsf{wp}\ \langle n; \mathsf{true}\rangle\ \{\Phi\} \qquad \triangleright\, \ell \mapsto^{[n]} v}{\mathsf{wp}\ \langle n; \mathsf{cas}(\ell, v, w)\rangle\ \{\Phi\}}$$

$$\textsc{wp-cas-fail}\ \frac{\ell \mapsto^{[n]} v' \mathrel{-\!\!*} \mathsf{wp}\ \langle n; \mathsf{false}\rangle\ \{\Phi\} \qquad \triangleright\, \ell \mapsto^{[n]} v' \qquad v \neq v'}{\mathsf{wp}\ \langle n; \mathsf{cas}(\ell, v, w)\rangle\ \{\Phi\}}$$

The propositions $\ell \mapsto^{[n]} v$ and $\ell \mapsto^{[n]}_q v$ are called points-to and fractional points-to propositions respectively. These are similar to their counterparts in other (concurrent) separation logics; they are simply annotated with the node they belong to. They both assert that a memory location ℓ in the heap of node n has value v. The former asserts full ownership of this location while the latter asserts only the ownership of a fraction $0 < q \leq 1$ of this memory location. In particular, we have $\ell \mapsto^{[n]}_1 v \dashv\vdash \ell \mapsto^{[n]} v$. The proposition $\mathit{IsNode}(n)$ indicates that the node n is a valid node in the system. It is required for allocation of new memory locations but not for reading and updating them. This is because having a (fractional) points-to proposition for a location on a node in and of itself is indicative of the fact that that node is a valid node.

$$\textsc{wp-socket}\ \frac{\forall z.\ z \overset{s}{\mapsto}^{[n]} \mathsf{None} \mathrel{-\!\!*} \mathsf{wp}\ \langle n; z\rangle\ \{\Phi\} \qquad \triangleright\, \mathit{IsNode}(n)}{\mathsf{wp}\ \langle n; \mathsf{socket}\rangle\ \{\Phi\}}$$

$$\textsc{wp-socket-bind-dyn}\ \frac{\begin{array}{l}\forall g.\ z \overset{s}{\mapsto}^{[n]} \mathsf{Some}\,(ip, p) * (ip, p) \overset{r}{\mapsto} g * (ip, p) \Rightarrow^{\mathsf{prot}} \phi \mathrel{-\!\!*} \mathsf{wp}\ \langle n; 0\rangle\ \{\Phi\} \\ \quad \mathit{freePorts}(ip, \{p\}) \qquad \overset{f}{\mapsto} A \qquad (ip, a) \notin A \qquad \phi \qquad z \overset{s}{\mapsto}^{[n]} \mathsf{None}\end{array}}{\mathsf{wp}\ \langle n; \mathsf{socketbind}\ z\,(ip, a)\rangle\ \{\Phi\}}$$

WP-SOCKET-BIND-STAT

$$\frac{\forall g.\ z \overset{s}{\mapsto}^{[n]} \text{Some}\,(ip,p) * (ip,p) \overset{r}{\mapsto} g \mathrel{-\!\!*} \text{wp}\,\langle n; 0\rangle\,\{\Phi\} \qquad \mathit{freePorts}(ip,\{p\}) \qquad \overset{f}{\mapsto} A \qquad (ip,p) \in A \qquad (ip,p) \Rightarrow^{\text{prot}} \phi \qquad z \overset{s}{\mapsto}^{[n]} \text{None}}{\text{wp}\,\langle n; \text{socketbind}\,z\,(ip,a)\rangle\,\{\Phi\}}$$

WP-SEND-TO-BOUND

$$\frac{\begin{array}{c}\forall m_{id}, M.\ \mathit{mSoup}(M) * m_{id} \overset{st}{\mapsto} (a,d,s) * P(a,d,s) \mathrel{\Rrightarrow\!\!*} \mathit{mSoup}(M) * \triangleright\, \phi(a,d,s) * Q(a,d,s) \\ z \overset{s}{\mapsto}^{[n]} \text{Some}\,(ip,p) \mathrel{-\!\!*} \text{wp}\,\langle n; \mathit{length}(s)\rangle\,\{\Phi\} \\ z \overset{s}{\mapsto}^{[n]} \text{Some}\,(ip,p) \qquad d \Rightarrow^{\text{prot}} \phi \qquad P \qquad Q\end{array}}{\text{wp}\,\langle n; \text{sendto}\,z\,s\,d\rangle\,\{\Phi\}}$$

WP-SEND-TO-UNBOUND

$$\frac{\begin{array}{c}\forall p, m_{id}, M.\ \mathit{mSoup}(M) * m_{id} \overset{st}{\mapsto} (a,d,s) * P(a,d,s) \mathrel{\Rrightarrow\!\!*} \mathit{mSoup}(M) * \triangleright\, \phi(a,d,s) * Q(a,d,s) \\ z \overset{s}{\mapsto}^{[n]} \text{None} \mathrel{-\!\!*} \text{wp}\,\langle n; \mathit{length}(s)\rangle\,\{\Phi\} \\ z \overset{s}{\mapsto}^{[n]} \text{None} \qquad d \Rightarrow^{\text{prot}} \phi \qquad \exists p.\ a = (ip,p) \qquad P \qquad Q \qquad \mathit{freePorts}(ip,\emptyset)\end{array}}{\text{wp}\,\langle n; \text{sendto}\,z\,s\,d\rangle\,\{\Phi\}}$$

WP-RECEIVE-FROM

$$\frac{\begin{array}{c}z \overset{s}{\mapsto}^{[n]} \text{Some}\,a * (\mathit{some} \vee \mathit{none}) \mathrel{-\!\!*} \text{wp}\,\langle n; r\rangle\,\{\Phi\} \\ \mathit{some} \triangleq \exists \mathit{mId}, m.\ r = \text{Some}\,(\mathit{body}(m), \mathit{from}(m)) * a \overset{r}{\mapsto} \{\mathit{mId} \mapsto m\} \cup g * m_{id} \overset{m}{\mapsto}_{\frac{3}{4}} m * \phi(\mathit{stable}(m)) \\ \mathit{none} \triangleq r = \text{None} * a \overset{r}{\mapsto} g \qquad z \overset{s}{\mapsto}^{[n]} \text{Some}\,a \qquad a \overset{r}{\mapsto} g \qquad a \Rightarrow^{\text{prot}} \phi\end{array}}{\text{wp}\,\langle n; \text{receivefrom}\,z\rangle\,\{\Phi\}}$$

Similar to allocation of locations on the heap, to allocate a socket by WP-SOCKET, one must provide $\mathit{IsNode}(n)$ to indicate that n is valid. A socket points to $z \overset{s}{\mapsto}^{[n]}$ None is returned, that enjoy the same properties as those for heaps.

We already touched upon binding of sockets to both primordial and dynamic addresses in 2, but here we present the actual rule as defined in Aneris. What is new is the ghost-assertion $(ip,p) \overset{r}{\mapsto} g$ which is a necessary evil if one wants to build protocols on top of the UDP communication channel. If we only used the $z \overset{s}{\mapsto}^{[n]}$ Some (ip,p) we could not prove that no other messages was received on the socket. Notice again, that the dynamic bind allows one to use a custom socket protocol ϕ.

When a socket has been bound to an address a, the duplicate assertion $a \Rightarrow^{\text{prot}} \phi$ is returned, stating, that all participants in the distributed system will obey the socket protocol.

Arguable, the most interesting part is the interaction with the socket protocols in sendto and receivefrom . For sending, one has to prove, via $\Rrightarrow\!\!*$, that the resources described by ϕ can be obtained from the resources one currently owns and the message stable resource $m_{id} \overset{st}{\mapsto} (a,d,s)$, intuitively saying that send operation took place. The P and Q are there to allow the client to prove custom protocols at the atomic place the send operation occurs. When complete, the resources guarding the socket protocol are transferred to the network while the messages is in transit.

UDP sockets allow for both sending messages through bound sockets and unbound sockets. For unbound sockets, an available socket will be picked during the atomic send and released immediately afterwards. To allow for picking an available port, the $\mathit{freePorts}(ip,\emptyset)$ has to be provided. Replies

on unbound ports is technically possible, however, logically one would know the socket protocol for the sender address.

When calling receivefrom one of two things can happen; either you will receive a message or no messages are available on the socket. If a message can be received, the resources described by ϕ are now transferred from the network to node owning the socket. Furthermore, a certificate that says the message has been received $m_{id} \overset{m}{\mapsto}_{\frac{3}{4}} m$ is also returned to the client. The fractional permission that the network keeps ensures that the client cannot change the state post-delivery.

It may not be completely obvious that one can encode protocols that require progress by these resources, however, consider the following socket protocol.

$$\phi(p)(m) \triangleq \exists a, m_{id}, n, \phi'.\, m_{id} \overset{st}{\mapsto} (a, p, n) * a \Rightarrow^{\text{prot}} \phi' * \\ (\forall m'_{id}.\, m_{id} \overset{m}{\mapsto}_{\frac{3}{4}} m * m'_{id} \overset{st}{\mapsto} (p, a, n+1) \mathrel{-\!\!*} \phi'(p, a, n+1))$$

Here, a socket can require that when another participant q sends a node on a socket to p, q must be able to receive a message where the number has been incremented by 1. However, if the protocol did not assert $m_{id} \overset{m}{\mapsto}_{\frac{3}{4}} m\ p$ could cheat and potentially send a message ahead of time. With the resources in place, such cheating can be avoided.

4.3 Adequacy

A (program) logic is at most as good as its soundness/adequacy theorem. That is, one needs to answer the question: "what do we get when prove a theorem in the logic?". Concretely in our case, what, if anything, can we conclude from a proof of a weakest precondition proven in Aneris' logic? The short answer is *safety*. That is, when running a distributed system for which we have proven a weakest precondition, none of the nodes in the system crash. This property, *i.e.*, weakest preconditions implying safety, is stronger than meets the eye at first. The reason for this is, that it is possible, that violating socket protocols can cause a node to crash, *e.g.*, a client of the lock server in 2.3. However, proving weakest precondition proves safety, so it must imply that protocols are followed for all parties in the distributed system.

The adequacy theorem. The statement of the adequacy theorem in our Coq development is the following:

```
Theorem adequacy `{distPreG Σ}
      (IPs : gset ip_address) (A : gset socket_address)
      e σ :
(∀ `{distG Σ}, (|={⊤}=> ∃ (f : socket_address → socket_interp Σ),
                  ↦^f A -★ ([★ set] a ∈ A, a ⇛^prot (f a)) -★
                   ([★ set] ip ∈ IPs, FreeIP ip) -★ WP e {{v, True }})%I) →
dom (gset ip_address) (state_ports_in_use σ) = IPs →
(∀ i, i ∈ IPs → state_ports_in_use σ !! i = Some ∅) →
state_heaps σ = ∅ → state_sockets σ = ∅ → state_lookup σ = ∅ → state_ms σ = ∅ →
safe e σ.
```

This theorem is a bit long but its reading is straightforward: It proves that running Aneris program e is safe starting from state σ if the following conditions hold:

- Under the assumption that resources are initialized, `distG Σ`, we can show that there are protocols such that having the given set of primordial sockets, `A`, with those protocols and

having free all necessary ip-addresses, `IPs`, we can show the weakest precondition for e in Iris. Notice that the `%I` instructs Coq to parse logical connectives as Iris connections instead of Coq connectives. The symbol `|={⊤}=>` is how we write the update modality in Coq where `⊤` is the mask of this modality for invariant names that we have omitted in this paper. We write `[⋆ set] a ∈ A` in Coq for the big separating conjunction connective.
- In σ there are no heaps, no sockets, and the lookup table and messages are also empty
- The set of IP-address in σ should be exactly `IPs` and the set of ports used in each of the ip-addresses should be empty.

The `distPreG Σ` is the assumption stating that the set of resources that Iris is parameterized with includes all necessary resources for initializing distributed systems. Note that individual proofs of some distributed systems might require more resources than basic resources for distributed systems for their verification.

The adequacy theorem above is a direct consequence of the adequacy theorem of Iris which says that closed (not relying on any particular resource) proofs of weakest preconditions imply safety. To prove the adequacy of Aneris, above, we need to show that all the resources necessary for distributed systems can be initialized appropriately. In other words, we need to instantiate `distG Σ`.

Note that the final result of the adequacy theorem above is safety, a fact within Coq independent of Aneris and Iris. In other words, when we verify a program within Aneris, we get that the program is safe to execute independent of Aneris or Iris. So indeed one must only trust Coq as formal system and need not trust Iris or our program logic Aneris build on top of it.

5 CASE STUDY: A LOAD BALANCER

As mentioned earlier, AnerisLang supports concurrency through its fork $\{e\}$ primitive. One example for illustrating the benefit of this primitive is server-side load balancing. Load balancing in general is used in order to distribute workload in a distributed system and is commonly utilized with the goal of achieving horizontal scaling wrt. providing Internet services.

5.1 Implementation of a Load Balancing Protocol

In the case of server-side load balancing, the distribution of work is done by a program listening on a port clients send their requests to. This program then sends the request to one of the available servers, waits for the answer from the server and sends this answer back to the client. In order to be able to handle requests from several clients simultaneously, the load balancer can employ concurrency by forking off a new thread for every available server in the system. Each of these threads will then listen for and handle requests Figure 7.

The `load_balancer` module expects an ip address, a port and a list of servers. It then creates and binds to a socket with the given ip address and port. Finally, it folds over the list of servers, forking off a new thread for each server, running the `serve` module with the newly-created socket, the given ip address, a fresh port number and the current server as arguments.

The `serve` module expects a socket, an ip address, a port and a server address. It first creates and binds to a new socket with the given ip address and port number. After this, `serve` tries to receive a message on the `load_balancer` socket. This message would be a request from a client. Once a request is received, it is passed on to the given server via the fresh socket and `serve` waits until it receives an answer from the server, which it finally passes on to the client via the `load_balancer` socket. This way the entire process is hidden from the client, whose view will be the same as if communicating with a single server handling its request.

```
rec load_balancer ip port servers :=
  let main := socket in
  socketbind skt (makeaddress ip port);
  list_fold
    (rec server acc :=
     fork (serve main ip acc server);
     acc + 1) 1100 servers

rec listen_wait skt :=
  match receivefrom skt with
    SOME m => m
  | NONE => listen_wait skt
  end
```

```
rec serve main ip port server :=
  let skt := socket in
  socketbind skt (makeaddress ip port);
  (rec loop :=
    match receivefrom main with
      SOME m =>
        let msg := Fst m in
        let sender := Snd m in
        let _ :=
          sendto skt msg server in
        let res :=
          Fst (listen_wait skt) in
        sendto main res sender;
        loop ()
    | NONE => loop ()
    end) ()
```

Fig. 7. An implementation of a load balancer in AnerisLang.

5.2 Specification and Protocols

In order to keep the specification of the load balancer as general as possible, we parameterize its socket protocol by two predicates $P_{in}, P_{out} : \mathit{MessageBody} \to \mathit{iProp}$ describing the client request and server response, respectively. These predicates may only depend on the body of the two messages, since the sender and receiver are changed when relaying the message forward from the load-balancer. The relay socket protocol is shown below:

$$\phi_{rel}\, P_{in} P_{out} \triangleq \lambda m.\, \exists \varphi.\, \mathit{from}(m) \Rightarrow^{\mathrm{prot}} \varphi * P_{in}(\mathit{body}(m)) * (\forall m'.\, P_{out}(\mathit{body}(m')) \mathrel{-\!\!*} \varphi(m'))$$

The load balancer and each of the servers are bound by the same socket protocol. Combined, this enables the `serve` module to relay requests from the `load_balancer` socket to the server and responses in the opposite direction without invalidating the socket protocol.

A client will have to show that his socket is bound to some protocol, that the body of the request fulfills the P_{in} predicate and that, given an answer whose body satisfies the P_{out} predicate, it is possible to fulfill the client protocol.

As mentioned above, all servers follows the same protocol. Any request the `serve` module receives already fulfills P_{in}, the module can safely relay the message and satisfy the protocol of the server it will contact. In order to satisfy the $\mathrel{-\!\!*}$ component, the `serve` module needs to be bound to a protocol, which will be satisfied by receiving an answer fulfilling P_{out}. This is easily achieved by defining the serve protocol as follows:

$$\phi_{serve} \triangleq \lambda m, P_{out}(\mathit{body}(m))$$

Since all instances of the `serve` module need access to the `load_balancer` socket in order to receive requests and send answers, we have to share the resources required for accessing a socket in an invariant.

$$\mathit{lb\_inv}\, n\, z\, a \triangleq \exists \gamma.\, \boxed{\exists g.\, z \overset{s}{\mapsto}^{[n]} \mathrm{Some}\, a * a \overset{r}{\mapsto} g}^{\mathcal{N}}$$

With these in place, the specification for the `load_balancer` module becomes: The specification for the `serve` module is:

$$\left\{\begin{array}{l} lb\_inv(n, main, ma) * ma \Longmapsto^{\text{prot}} \phi_{rel} P_{in} P_{out} * server \Longmapsto^{\text{prot}} \phi_{rel} P_{in} P_{out} * \overset{f}{\mapsto} (A) * \\ (ip, p) \notin A * freeIp(ip) \end{array}\right\}$$

$\langle n;$ serve main ip port server$\rangle$

{True}

This specification is fairly straightforward. It requires the given address for the correspoding `load_balancer` module and the server to be bound to the relay protocol and the invariant to be established. Moreover, the address matching the given ip address and port should not be in use already and the given port should be free on this ip address.

6 CASE STUDY: TWO-PHASE COMMIT AND REPLICATED LOGGING

For transaction processing and in databases the problem of (distributed) commit is as follows; one operation should be performed by all participants in the system or none at all. A simplistic solution to distributed commit, called the *one-phase* commit, would broadcast to all participants to commit or abandon a transaction. However, in distributed systems it is often the case that not all parties can commit due to other constraints - or a node could simply fail.

The *two-phase commit* protocol (TPC), due to Gray [1978], is a distributed consensus protocol that coordinates all participants to commit or abort. It is widely used in practice because it is somewhat resilient to a variety of failures, such as unreliable message delivery and transient participant crashes.

The reason for studying this protocol in Aneris is: (1) it is widely used in the real-world, (2) it is a complex network protocol thus serves as a decent benchmark for reasoning in Aneris with the Aneris tailored proofmode, and (3) the implementation should consist of network-capable modules that could be used abstractly by clients, without relying on the specific implementation.

6.1 Implementation of the Two-Phase Commit Protocol

The two-phase commit protocol consists of the following two phases, each consisting of two steps:

(1)(a) The coordinator sends out a vote-request to each participant.
(b) A participant that receives a vote-request, replies with either vote-commit to inform that it is prepared to locally commit or a vote to abort.
(2)(a) The coordinator collects all votes and determines a result. If all participants voted commit, the coordinator sends a global-commit to all. Otherwise, at least one participant voted abort and the coordinator sends global-abort to all.
(b) All participants that voted for a commit wait for the final verdict from the coordinator. If the participant receives a global-commit it locally commits the transaction, otherwise the transaction is locally aborted. All must reply ACK.

These steps are shown as finite state machines in Figure 8 and a TPC-module that satisfies the conceptual description is shown in Figure 9. Our abstract model differs a bit from the standard diagram ([Tanenbaum and Van Steen 2007]) because we reuse the same code and sockets for communication between coordinators and participants. Every state is therefore tagged with a unique round number and each participant remembers the previous result (initially, COMMIT suffices). The dashed arrows are internal updates of the state, to allow for reuse of the protocol.

The `tpc_coordinator` module expects an initial request message to be provided, along with a bound socket, a list of participants and a function to make a decision when all votes have been received. Internally, it uses two local references; one to collect all the votes and one to count the number of acknowledgements.

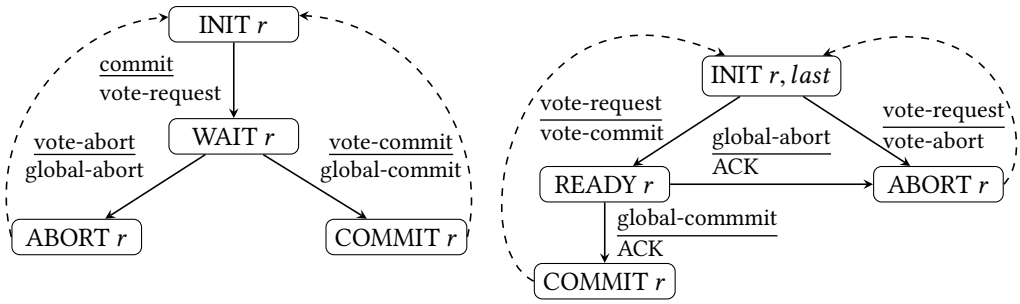

Fig. 8. The finite state machine for the coordinator in TPC on the left and for the participant on the right.

```
rec tpc_coordinate m skt ps dec :=
  let count := list_length ps in
  let msgs := ref (list_make ()) in
  let ack := ref 0 in
  list_iter (rec n := sendto skt m n) ps;
  listen skt (rec handler m from :=
    let msgs' := !msgs in
    msgs ← list_cons m msgs';
    if: list_length !msgs = count
    then () else listen skt handler);
  let res := dec !msgs in
  list_iter (rec n := sendto skt res n) ps;
  listen skt (rec handler m from :=
    ack ← !ack + 1;
    if !ack = count then res
    else listen skt handler)
```

```
rec tpc_participant skt vote fin :=
  let msg := listen_wait skt in
  let act := vote (fst msg) in
  sendto skt act (snd msg);
  let res := listen_wait skt in
  fin (fst res);
  sendto skt "ACK" (snd res);
  tpc_participant skt req fin

rec listen_wait skt :=
  match receivefrom skt with
    SOME m => m
  | NONE => listen_wait skt
  end
```

Fig. 9. An implementation of the two-phase commit in AnerisLang. The functions list_make, list_cons and list_length are library utility functions for operations on lists implemented as splines.

The `tpc_participant` module expects a socket and two handlers - one handler to decide on a vote and one handler to finalize on the decision made by the coordinator. When invoked, the module listens for incoming requests, decides on a vote and waits for a global decision from the coordinator. One could argue that `tpc_participant` is not faithfully implemented according to the TPC state-transition system because it always blocks until a decision is made by the coordinator. However, because each node can employ concurrency, the client can decide on concurrent work, in particular it can engage in other rounds of TPC with other coordinators. Notice as well that there are no round numbers in the implementation - these are purely in the abstract model to strengthen the specification.

6.2 Specification and Protocols

The approach for verifying programs in Aneris is similar to the approach in Iris - one has to consider the resources needed that correctly capture the intended purpose. We will use the following pre-created resources for a TPC instance, having a coordinator c and participants $p \in ps$:

- $parts(ps)$: Keeps track of all participants for a concrete TPC instance and is duplicable and unmodifiable. Conceptually, it fixes the participants in the TPC instance.

- $p \overset{c}{\mapsto} (r, CS)$: A participant-specific assertion on the state of the coordinator CS (as seen in Figure 8) for round r. The coordinator c owns an assertion regarding its own state $c \overset{c}{\mapsto} (r, CS)$. We require that $\forall a \in ps \cup \{c\}, a \overset{c}{\mapsto} (r, SC)$ are in agreement for all r and CS. Technically, this is stated as an invariant *tpc_inv*.
- $p \overset{p}{\mapsto}\{\pi\}\,(r, PS)$: An assertion for each participant p regarding its own state. The resource can be split arbitrarily $p \overset{p}{\mapsto}\{\pi\}\,(r, PS) \dashv\vdash p \overset{p}{\mapsto}\{\pi_1\}\,(r, PS) * p \overset{p}{\mapsto}\{\pi_2\}\,(r, PS)$ as long as $\pi_1 + \pi_2 = \pi \leq 1$ and $\pi_i > 0$. For any two fractions $p \overset{p}{\mapsto}\{\pi_1\}\,x * p \overset{p}{\mapsto}\{\pi_2\}\,x'$ we have $x = x'$. Finally, if one owns all fractions $p \overset{p}{\mapsto}\{1\}\,(r, PS)$, the points to can be freely updated since the environment cannot own any fractions.

To allow the client to use the TPC protocol in whatever fashion it wishes, the implementation is parameterized at all places dealing with messages. Consequently, when proving a client, the prover has to provide the decidable predicates *is_req*, *is_vote*, *is_abort* and *is_global* of type $(String \times \mathbb{N}) \to$ Prop. The client is free to pick $P : (Address \times String) \to iProp$ and $Q : (Address \times \mathbb{N}) \to iProp$, conceptually, the local pre- and post-condition for each participant. The socket protocol for the coordinator is shown below.

$$\phi_{vote} \triangleq \lambda m, \exists s, r, ps, sp.\, from(m) = s * parts(\{s\} \cup ps) * is\_vote(body(m), r) * s \overset{c}{\mapsto} (r, \textsc{wait}) * (is\_abort(body(m), r) * s \overset{p}{\mapsto}\{\tfrac{3}{4}\}\,(r, \textsc{abort}) \vee \neg is\_abort(body(m), r) * s \overset{p}{\mapsto}\{\tfrac{3}{4}\}\,(r, \textsc{commit}))$$

$$\phi_{ack} \triangleq \lambda m, \exists m', cs, ps, pr, r, s.\, s = from(m) * parts(\{s\} \cup ps) * s \overset{c}{\mapsto} (r, cs) * s \overset{p}{\mapsto}\{\tfrac{3}{4}\}\,(r, \textsc{init}\, pr) * (cs = \textsc{commit} * pr = \textsc{commit} * Q(s, r) \vee cs = \textsc{abort} * pr = \textsc{abort} * P(m', s))$$

$$\phi_{coord} \triangleq \lambda m, \phi_{vote}(m) \vee \phi_{ack}(m)$$

For a participant p to send a vote to the coordinator c, it has to show that it is indeed a participant $parts(\{s\} \cup ps)$, that the message is a vote for that round and that it owns the resources for the state of c, $s \overset{c}{\mapsto} (r, \textsc{wait})$, and a resource that confirms that the state of p matches p's vote. For the participant to send an acknowledgement, it has to prove it transitioned to INIT by sending $s \overset{p}{\mapsto}\{\tfrac{3}{4}\}\,(r, \textsc{init}\, pr)$ the state pr should match the global decision. If the decision was to commit, the participant should send the updated resources for Q, otherwise it should return the resources described by P. The socket protocol for the participant is as follows:

$$\phi_{rec}(p) \triangleq \lambda m, \exists ps, r, sp.\, parts(\{p\} \cup ps) * is\_req(body(m), r + 1) * from(m) \Rightarrow^{\text{prot}} \phi_{coord} * p \overset{c}{\mapsto} (r + 1, \textsc{wait}) * p \overset{p}{\mapsto}\{\tfrac{3}{4}\}\,(r, \textsc{init}\, sp)$$

$$\phi_{glob}(p) \triangleq \lambda m, \exists ga, ms, ps, r, sc, sp.\, \{from(m) | m \in ms\} = ps * is\_global(body(m), r) * ga = \{m | m \in ms \wedge is\_abort(m, r)\} * parts(ps) * from(m) \Rightarrow^{\text{prot}} \phi_{coord} * p \overset{c}{\mapsto} (r, sc) * p \overset{p}{\mapsto}\{\tfrac{3}{4}\}\,(r, sp) * \left(\ast_{m \in ms} \exists m_{id}, \pi.\, is\_vote(body(ms), r) * m_{id} \overset{m}{\mapsto}\{\pi\} m \right) * (ga = \emptyset \wedge \neg is\_abort(body(m), r) \wedge sc = \textsc{commit} \vee (ga \neq \emptyset \wedge is\_abort(body(m), r) \wedge sc = \textsc{abort})$$

$$\phi_{part}(p) \triangleq \lambda m, \phi_{rec}(p)(m) \vee \phi_{glob}(p)(m)$$

In order to send a request for a round $r + 1$ of TPC to a particioant p, a coordinator has to prove that it is in fact a valid request and that the address the participant will reply on is bound to a coordinator protocol ($from(m) \Rightarrow^{\text{prot}} \phi_{coord}$). Furthermore, the coordinator has to show it is in the WAIT state

by transferring $p \overset{c}{\mapsto} (r+1, \text{WAIT})$ and give up $\overset{p}{\mapsto}\{\frac{3}{4}\}\,(r, \text{INIT}\ sp)$ to allow the participant to make a transition. Finally, the coordinator can broadcast a global decision to each participant when it has received all messages for a round. This is guaranteed by $(\ast_{m \in ms}\, \exists m_{id}, \pi.\ is\_vote(body(ms), r) * m_{id} \overset{m}{\mapsto}\{\pi\}m)$, where $\ast$ is iterated separating conjunction over finite sets. The coordinator also has to be honest in its decision, thus if any participant replied with an abort message, the global message should be abort as well and the final state of the coordinator should be ABORT. Notice as well, that for each message to a participant, the coordinator will pass in the assertion $(from(m) \Rightarrow^{\text{prot}} \phi_{coord})$. Therefore, the participant does not need to have prior knowledge of the coordinator - it can even change from round to round.

With the TPC protocol in place, we can finally give a specification to the two TPC modules.

$$\left\{\begin{array}{l} tpc\_inv * parts(ps) * z_n \mapsto \text{Some}\, p * p \Rightarrow^{\text{prot}} \phi_{part}(p) * p \overset{p}{\mapsto}\{\frac{1}{4}\}\,(r, \text{INIT}\ sp) * \\ is\_reqf(req) * is\_finf(fin) \end{array}\right\}$$
$$\langle n; \text{tpc\_participant z req fin}\rangle$$
$$\{\text{True}\}$$

The tpc_participant specification is straightforward. It requires ownership of a bound socket guarded by a participant protocol $\phi_{part}(p)$ and fractional ownership of its own state, initialized to be INIT. The specification for tpc_coordinate is a bit more involved:

$$\left\{\begin{array}{l} ps \equiv psV * is\_req(m, r+1) * tpc\_inv * parts(ps) * z_n \mapsto \text{Some}\, a * a \Rightarrow^{\text{prot}} \phi_{coord} * a \overset{c}{\mapsto} (r, \text{INIT}) * \\ is\_decf(dec) * \ast_{p \in ps}\, \exists sp, p \Rightarrow^{\text{prot}} \phi_{part}(p) * p \overset{c}{\mapsto} (r, \text{INIT}) * p \overset{p}{\mapsto}\{\frac{3}{4}\}\,(r, \text{INIT}\ sp) * P(p, msg) \end{array}\right\}$$
$$\langle n; \text{tpc\_coordinate m z psV dec}\rangle$$
$$\left\{\begin{array}{l} \langle n; v\rangle.\ \exists sc, sp.\ is\_global(v, r+1) * z_n \mapsto \text{Some}\, a * a \overset{c}{\mapsto} (r+1, sc) * \\ \qquad \left(\ast_{p \in ps}\, p \overset{c}{\mapsto} (r, sc) * p \overset{p}{\mapsto}\{\frac{3}{4}\}\,(r, \text{INIT}\ sp)\right) * \\ \qquad (is\_abort(v, r+1) * sc = \text{ABORT} * sp = \text{ABORT} * \ast_{p \in ps}\, \exists m.\ P(p, m) \vee \\ \qquad\quad \neg is\_abort(v, r+1) * sc = \text{COMMIT} * sp = \text{COMMIT} * \ast_{p \in ps}\, Q(p, r)) \end{array}\right\}$$

To call tpc_coordinate, one has to pass ownership of a socket z_n already bound to some address guarded by the ϕ_{coord} protocol. The list of nodes psV should be "equivalent" to the set of participants and for each participant, besides the resources describing the participant's view of the coordinators and participant's state-transition system, we also need knowledge about the participant address being guarded by a suitable protocol $\phi_{part}(p)$.

The post-condition here is the most exciting part, mainly because it is exactly what one would expect. Either all participants, along with the coordinator agreed to commit to which we obtain $Q(p, r)$ for each participant or they all agreed to abort, to which we get back $P(p, m)$. We elide the specifications *is_decf*, *is_reqf* and *is_finf* to the Coq development.

As to reap the fruits of our hard labor, we will show a client that use the TPC modules.

6.3 Replicated Logging

As noted in Sergey et al. [2017], clients of core consensus protocols have not received much focus from other major verification efforts ([Hawblitzel et al. 2015; Rahli et al. 2015; Wilcox et al. 2015]) with the exception of ([Lesani et al. 2016; Sergey et al. 2017]). In the work by Sergey et al. [2017],

```
rec logger log addr m dbs :=
  let skt := socket() in
  let dec := rec msgs :=
    let r = list_fold (rec a m :=
      a && m = "COMMIT") true msgs in
    if r then "COMMIT" else "ABORT" in
  socketbind skt addr;
  tpc_coordinate ("REQUEST_" ^ m)
                 skt dbs dec
```

```
rec db addr :=
  let skt := socket() in
  let wait := ref "" in
  let log := ref "" in
  let req := rec m := wait ← val_of m;
                      "COMMIT" in
  let fin := rec m := if m = "COMMIT"
             then log ← !log ^ !wait
             else () in
  socketbind skt addr;
  tpc_participant skt req fin
```

Fig. 10. Replicated logging that use two-phase commit modules in AnerisLang. ^ is string concatenation.

both an implementation of TPC and a client for replicated logging with a side-channel for error recovery are shown. We will prove a similar client shown in Figure 10, without a side-channel, to allow for comparison with our verification efforts.

The replicated `logger` opens a socket `skt` on address `addr` and initiates a TPC-round for `dbs` by sending `"REQUEST_" ^ msg`. The decision handler `dec` is called by the TPC module when all votes have been received.

On the side of the database, `db`, we use an internal reference `log` as the log. [6] Upon an incoming request, the message is parsed to get out the value (`val_of m`) and the log to append is stored in `wait`. If the global decision by `logger` is "COMMIT", the string stored in `wait` will be appended to the `log`.

To logically describe the local state of each database we use the following heap-like mapsto predicates, $p \overset{l}{\mapsto}\{\pi\}\ log$ and $p \overset{w}{\mapsto}\{\pi\}\ log, wait$, that keep track of the log and waiting commit for each participant p. The predicate P and Q, which we instantiate TPC with are defined below:

$$P \triangleq \lambda p, m.\, \exists log, s.\, m = "REQUEST\_" \mathbin{@} s * p \overset{l}{\mapsto}\{\tfrac{1}{2}\}\ log * p \overset{w}{\mapsto}\{\tfrac{1}{4}\}\ log, s$$

$$Q \triangleq \lambda p, n.\, \exists log, s.\, p \overset{l}{\mapsto}\{\tfrac{1}{2}\}\ log @ s * p \overset{w}{\mapsto}\{\tfrac{1}{4}\}\ log, s$$

With these resources in place, we can give the `logger` the following specification:

$$\left\{ \begin{array}{l} tpc\_inv * parts(dbs) * freePorts(ip(addr), \{port(addr)\}) * is\_req(m) \\ \quad \ast_{p \in dbs}\ \exists sp, p \Rightarrow^{\mathsf{prot}} \phi_{part}(p) * p \overset{c}{\mapsto} (r, \mathsf{INIT}) * p \overset{p}{\mapsto}\{\tfrac{3}{4}\}\ (r, \mathsf{INIT}\ sp) * P(p, m) \end{array} \right\}$$
$$\langle n; logger\ log\ addr\ dbs \rangle$$
$$\left\{ \begin{array}{l} \langle n; v \rangle.\, \exists m, r.\ \ast_{p \in dbs}\ \exists sp.\, p \overset{c}{\mapsto} (r, \mathsf{INIT}) * p \overset{p}{\mapsto}\{\tfrac{3}{4}\}\ (r, \mathsf{INIT}\ sp) * \\ \qquad \left(v = "COMMIT" * \ast_{p \in dbs}\ Q(p, r) \vee v = "ABORT" * \ast_{p \in dbs}\ P(p, m) \right) \end{array} \right\}$$

The proof follows directly, in a modular node-local fashion, from applying the specification of `tpc_coordinate`.

It seems unlikely that the consensus-protocol could be swapped with other consensus protocols, such as Raft or Paxos without also changing the proof, even if `logger` was parameterized

[6] Ideally, this should be "persistent" storage, however, to keep the example concrete, we use a local reference

by a protocol. Raft and Paxos require only a majority of participants for committing, which is observationally different from TPC.

7 RELATED WORK

Verification of distributed systems has received a fair amount of attention. In order to give a better overview, we have divided related work into four categories.

7.1 Model-Checking of Distributed Protocols

Previous work on verification of distributed systems has traditionally focused on verification of protocols or core network components by means of model-checking. Frameworks for showing safety and liveness properties, such as SPIN Holzmann [1997], and TLA+ [Lamport 1993], have had great success. A clear benefit of using model-checking frameworks is that they allow to state both safety and liveness assertions as LTL assertions [Pnueli 1977]. Mace [Killian et al. 2007] provides a suite for building and model-checking distributed systems with asynchronous protocols, including liveness conditions. Chapar [Lesani et al. 2016] allows for model-checking of programs that use causally consistent distributed key-value stores. Neither of these languages provide higher-order functions or thread-based concurrency. Additionally, model-checking frameworks cannot prove the absence of errors in general, they can only show it for a specific model.

7.2 Session Types for Giving Types to Protocols

Session types have been studied for a wide range of process calculi, in particular, typed π-calculus. The original idea was to describe two-party communication protocols as a type to ensure communication safety and progress [Honda et al. 1998]. This was later extended to multi-party asynchronous channels [Honda et al. 2008], multi-role [Deniélou and Yoshida 2011] informally modeling topics of actor-based message-passing and to dependent session types allowing quantification over messages [Toninho et al. 2011]. Our socket protocol definitions are quite similar to the multi-party asynchronous session types in the sense that our sockets are multi-party as well and progress can be encoded by having suitable ghost-assertions and using magic-wand (SEE REF TO MESSAGES)

7.3 Distributed Hoare Type Theory and Concurrent Program Logics

Disel [Sergey et al. 2017] is a Hoare Type Theory for distributed program verification in Coq with ideas from separation logic. It provides the novel protocol-tailored rules WithInv and Frame which allow for modularity of proofs under the condition of an inductive invariant and distributed systems composition. We obtain composition by node-local reasoning, which is possible because we work in a logic that requires us to state stable assertions on the environment locally.

Disel cannot hide mutable state in the system and it must be made known to all in a protocol. Additionally, node-local mutable state can only be altered upon send/receive. Besides having thread-based concurrency on nodes, AnerisLang also provides "proper" local mutable state that can be hidden, which the authors of Disel described as interesting future work.

However, in Disel, programs can be extracted into runnable OCaml programs, which is on our agenda for future work.

IronFleet [Hawblitzel et al. 2015] allows for building provably correct distributed systems by combining TLA-style state-machine refinement with Hoare-logic verification in a layered approach, all embedded in Dafny [Leino 2010]. IronFleet even allows for liveness assertions. The top layer is a simple specification of the system's behavior, the bottom layer is the actual implementation, and each layer is proven to satisfy the layer above it. The verification results of IronFleet are impressive, however, it seems like composition of protocols requires progressing any changes through the

entire stack of layers. Compared to IronFleet, verified components in Aneris are easier to compose and can employ thread-local reasoning.

The concurrent program logic closest to our work is naturally Krebbers et al. [2017a], since it is the foundation we have based Aneris upon. Other concurrent program logics, [Dinsdale-Young et al. 2013; Nanevski et al. 2014; O'hearn 2007; Svendsen and Birkedal 2014; Turon et al. 2014] to name a few, all have some notion of thread-local reasoning or rely-guarantee reasoning. We believe we have successfully lifted that principle to reason about individual nodes locally in distributed systems.

7.4 Other Distributed Verification Efforts

Verdi [Wilcox et al. 2015] is a framework for writing and verifying implementations of distributed algorithms in Coq, providing a novel approach to network semantics and fault models. To achieve compositionality, the authors introduced *verified system transformers*, that is, a function that transforms one implementation to another implementation, which has different assumptions about its environment. This makes vertical composition difficult for clients of proven protocols and the language feature set is more expressive.

EventML [Rahli et al. 2015, 2017] is a functional language in the ML family that can be used for coding distributed protocols using high-level combinators from the Logic of Events, and verify them in the Nuprl interactive theorem prover. It is not quite clear how modular reasoning works, since one works within the model, however, the notion of a central main observer is akin to our system node.

8 CONCLUSION AND FUTURE WORK

Distributed systems are quite ubiquitous nowadays and hence it is essential to be able to verify them. In this paper we presented Aneris a framework for writing and verifying distributed systems in Coq on top of the framework of the Iris program logic. From a programming point of view, the important aspect of Aneris is that it is quite feature rich: it is basically a concurrent ML-like programming language with network primitives. This allows individual nodes to internally use higher-order heap and concurrency to write efficient programs.

On the program logic side the Aneris framework provides node-local reasoning. That is, we can reason about individual nodes in isolation as we do reason about individual threads in each node in isolation in what is known as thread-local reasoning. We demonstrated versatility of Aneris by verifying interesting distributed systems implemented and verified within Aneris. The adequacy theorem of Aneris implies that these programs are safe to run.

As of writing, AnerisLang is suitable for verifying code written in other operationally similar languages, such as OCaml, by writing it in the DSL defined in Coq. However, because the language is so realistic, one could write a simple transpiler for a *one-to-one* translation of terms in AnerisLang to terms in OCaml, without the need for code-extraction. A transpiler would make it feasible to use AnerisLang for a verify-first approach.

In fact, for an earlier version of AnerisLang, we had an existing compiler with a small standard library consisting of 15 lines of OCaml, mainly for parsing received bytes from sockets to strings.

REFERENCES

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *ACM SIGPLAN Notices*, Vol. 44. ACM, 340–353.

Andrew Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *TOPLAS* 23, 5 (2001), 657–683.

Andrew Appel, Paul-André Melliès, Christopher Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *POPL*.

Pierre-Malo Deniélou and Nobuko Yoshida. 2011. Dynamic multirole session types. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 435–446.

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 287–300.

D. Dreyer, A. Ahmed, and L. Birkedal. 2011. Logical Step-Indexed Logical Relations. *LMCS* 7, 2:16 (2011).

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010. The impact of higher-order state and control effects on local relational reasoning. In *ACM Sigplan Notices*, Vol. 45. ACM, 143–156.

Robert W Floyd. 1967. Assigning meanings to programs. *Mathematical aspects of computer science* 19, 19-32 (1967), 1.

James N Gray. 1978. Notes on data base operating systems. In *Operating Systems*. Springer, 393–481.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 1–17.

Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.

Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*. Springer, 122–138.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. *ACM SIGPLAN Notices* 43, 1 (2008), 273–284.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 66.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650.

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Charles Edwin Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin M Vahdat. 2007. Mace: language support for building distributed systems. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 179–188.

Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In *European Symposium on Programming (ESOP)*.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. *ACM SIGPLAN Notices* 52, 1 (2017), 205–217.

Leslie Lamport. 1977. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering* 2 (1977), 125–143.

Leslie Lamport. 1978. The implementation of reliable distributed multiprocess systems. *Computer networks* 2, 2 (1978), 95–114.

Leslie Lamport. 1993. Hybrid systems in TLA+. In *Hybrid Systems*. Springer, 77–102.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

Mohsen Lesani, Christian J Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 357–370.

Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. 2014. Communicating state transition systems for fine-grained concurrent resources. In *European Symposium on Programming Languages and Systems*. Springer, 290–310.

Peter W O'hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical computer science* 375, 1-3 (2007), 271–307.

Amir Pnueli. 1977. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*. IEEE, 46–57.

Vincent Rahli, David Guaspari, Mark Bickford, and Robert L Constable. 2015. Formal specification, verification, and implementation of fault-tolerant systems using EventML. *Electronic Communications of the EASST* 72 (2015).

Vincent Rahli, David Guaspari, Mark Bickford, and Robert L Constable. 2017. EventML: Specification, verification, and implementation of crash-tolerant state machine replication systems. *Science of Computer Programming* 148 (2017), 26–48.

John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE, 55–74.

Ilya Sergey, James R Wilcox, and Zachary Tatlock. 2017. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 28.

Kasper Svendsen and Lars Birkedal. 2014. Impredicative concurrent abstract predicates. In *European Symposium on Programming Languages and Systems*. Springer, 149–168.

Andrew S Tanenbaum and Maarten Van Steen. 2007. *Distributed systems: principles and paradigms*. Prentice-Hall.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2017. A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 64.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2011. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*. ACM, 161–172.

Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. 2014. GPS: Navigating weak memory with ghosts, protocols, and separation. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 691–707.

James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 357–368.