Verifying Liveness Properties of Distributed Systems via Trace Refinement in Higher-Order Concurrent Separation Logic

JONAS KASTBERG HINRICHSEN[*], Aarhus University, Denmark
LÉO STEFANESCO[*], MPI-SWS, Germany
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

Distributed systems are notoriously difficult to design, implement, and reason about. This is especially true for liveness properties which state that a desired state of the system is guaranteed to eventually be reached. A lot of complexity is found in systems' implementations as these often employ features such as higher-order programming features and concurrency which are difficult to formally reason about even without the added complexity introduced by the asynchronous communication present in distributed systems, or considering liveness properties. In this paper we present Fairneris, the first higher-order concurrent program logic for formal and foundational verification of both safety and *liveness* properties of distributed systems' implementations. This work builds on the recent success of the Trillium program logic framework for refinement-based reasoning about higher-order concurrent programs, including liveness reasoning, and that of the Aneris program logic for reasoning about (only safety properties) of distributed systems. To demonstrate the capabilities of Fairneris we verify both safety and liveness properties of an implementation of the Stenning sliding window reliable transfer protocol on top of UDP-like unreliable networking primitives. Results presented in this paper are mechanized on top of the Coq proof assistant.

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

1 Introduction

Distributed systems are a natural target for formal verification, as they are notoriously difficult to get right and to test exhaustively. Moreover, they often serve critical functions. Traditionally, most works on studying correctness of distributed systems have focused on reasoning about correctness of communication protocols, *e.g.*, using model checking tools like SPIN [17] and TLA+ [22]. However, implementations of distributed systems often involve details not specified in high-level specifications [7]. Hence, it is important to verify correctness of the implementation of distributed systems.

Over the past decade many advances have been made in this direction, *e.g.*, Aneris [20], Disel [30], Grove [31], Igloo [32], IronFleet [16], and Verdi [35]. Virtually all these systems only support reasoning about so-called *safety* [21] properties, *i.e.*, properties that state nothing bad ever happens, *e.g.*, "the server never sends an invalid response". The only exception is IronFleet which also supports reasoning about *liveness* [21] properties, *i.e.*, properties that state some desired state is eventually reached, *e.g.*, "the server responds to all requests". These two families of properties, safety and liveness, are complete in the sense that any property is the conjunction of a safety property and a liveness property [3]. IronFleet manages to prove liveness by reducing it to solving termination of sequential programs. In IronFleet each node of the distributed system must be written as a single-threaded event loop. The high-level argument of the liveness proof of IronFleet is that distributed systems implemented in IronFleet are live whenever all event handlers terminate, because the event loop goes though actions in a round robin fashion — this is in turn possible

[*]Equal contribution.

Authors' Contact Information: Jonas Kastberg Hinrichsen, Aarhus University, , Denmark, hinrichsen@cs.au.dk; Léo Stefanesco, MPI-SWS, , Germany, lstefane@mpi-sws.org; Amin Timany, Aarhus University, , Denmark, timany@cs.au.dk; Lars Birkedal, Aarhus University, , Denmark, birkedal@cs.au.dk.

because all actions are always enabled as each handler must first check if its preconditions hold, and if not do nothing. As pointed out by Chandra et al. [7], it is important for distributed systems' nodes to run concurrent threads. In addition to concurrency, higher-order programming features such as higher-order functions, objects in object-oriented languages, *etc.*, are also crucial for engineering distributed software [6].

The key reason for the success of all these systems, Aneris [20], Disel [30], Grove [31], Igloo [32], IronFleet [16], and Verdi [35], is *modularity*. That is, in all these systems each module (function, method, class, thread, *etc.*), is proven correct against its own specification in isolation from other modules, under the assumption that the other modules are correct with respect to their own specification. Modular reasoning about liveness properties of concurrent programs is notoriously difficult [10, 34] (and hence also for distributed programs, even when each node is single-threaded). This is only exacerbated by higher-order programming features. In fact among the state of the art program logics for reasoning about liveness of concurrent programs, Lili [23], TaDa-live [10], and Fairis [34], only Fairis supports higher-order programming features.

1.1 The Fairneris Program Logic

In this paper we present Fairneris, the first program logic for *foundational* reasoning about both safety and *liveness* properties of arbitrary *higher-order* and *concurrent* programs that have access to network sockets. That is, not just programs with a fixed main event loop as in IronFleet [16] or Verdi [35]. By foundational we mean that everything is checked by a proof assistant; the Coq proof assistant in this case. Fairneris is built on top of two existing program logics: Fairis [34] and Aneris [20] — hence, the suggestive name Fairneris. Fairis [34] is the state-of-the-art foundational program logic based on the Trillium framework [34] which is in turn based on the Iris framework [19] for modular reasoning about liveness of concurrent programs. Aneris [20] is the state-of-the-art foundational program logic, also based on the Iris framework [19], for reasoning about safety properties of higher-order and concurrent distributed systems. We note that the motivation for this choice is beyond the mere convenience that both program logics are based on the Iris framework. As mentioned earlier, Fairis is the only program logic for reasoning about liveness of higher-order concurrent programs. Furthermore, the only program logics other than Aneris that support reasoning about higher-order concurrent distributed systems are Grove [31] and Igloo [32]. As we will explain in §6, it is not clear how the extend the methodology of Igloo to reasoning about liveness properties, at least not by following a methodology similar to that of Fairis. As for Grove, its methodology is very close to that of Aneris, it is also based on the Iris framework. However, Grove focuses on verifying crash recovery and reconfiguration of distributed systems.

1.2 Challenges

Unreliability of the Network. In the case of distributed systems, liveness is even more delicate than for shared memory concurrency, because the *unreliability* of the network also needs to be taken into account. This is a subtle question: on the one hand, a totally reliable network is not a realistic assumption. On the other hand, it is impossible to have non-trivial live distributed systems under a completely unreliable network [13]. Thus, we make the standard choice of assuming fairness of the network, *i.e.*, that the network satisfies the *eventual delivery* property: a message sent infinitely often is eventually delivered. In other words, we do not consider systems where a pair of nodes is disconnected forever. However, we do assume that messages can be dropped (though no indefinitely), reordered, or duplicated. Hence, many distributed programs have to retransmit messages and acknowledge received messages. Moreover, following and extending Aneris, in addition to unreliability, we work with a realistic model of network programming in the sense that packets are delivered by the network to the recipient node by being appended to the recipient socket's FIFO buffer. Thus, the program must repeatedly use the **recv** operation to retrieve newer network messages. As a result, the unreliability of the network greatly complicates reasoning about liveness. In fact, to reason about liveness one must repeatedly switch between reasoning about the program and reasoning about the network: the reason that the communication can progress to sending some message m_{k+1} is that message m_k was received (and acknowledged, if necessary) which can only happen if the thread calling **recv** is scheduled (depends on fairness of scheduling) after the network delivers m_k (depends on network's fairness) because the thread sending m_k was scheduled enough times (depends on fairness of scheduling), *etc.* IronFleet sidesteps this problem by working with single-threaded programs, and by considering retransmissions as events handled by the node's fixed event loop.

Network Abstraction. Arguably, the raison d'être of a program logic is to simplify reasoning by abstracting (hiding) the less relevant details that can/should be treated once and for all by the program logic for all programs, *e.g.*, when reasoning about (safety or liveness) of concurrent programs, we do not consider the internals of the memory allocator, different possible schedulings, or the state of the scheduler. Similarly, as a good program logic, Fairneris should, *and indeed does*, hide the details of the network like eventual delivery, details of the thread scheduler, and their interaction. One of the core challenges of our work is to abstract away the details of the network when combining Fairis and Aneris. The approach of Fairis to verifying liveness properties is via establishing a refinement relation between the program e and the user-picked high-level model $\mathcal{M}$, expressed as a labeled transition system (LTS), which intuitively captures the essence of the computation of the program. More precisely, the program logic of Fairis, allows its users to prove a refinement relation between the program e and the user-picked high-level model $\mathcal{M}$, under a user-picked relation ξ reflecting how the state of the program must relate to the state of the model. A theorem, proven once and for all about the Fairis refinement relations, states that for any fair (fairly scheduled) trace *ex* of the program there is a corresponding fair trace *utr* of the model such that whenever the trace *utr* satisfies a liveness property P, so does *ex*. Thus, using the methodology of Fairis (and similarly Fairneris), to prove a liveness property P for a program e, one only needs to show that there is a model $\mathcal{M}$ related to e that satisfies P (proven via the program logic of Fairis).

One obvious way to combine Fairis and Aneris is to reflect the entire state of the network in the model. This is indeed what Timany et al. [34] do to obtain a version of Aneris which can take advantage of refinement-based reasoning. However, they use this approach to show that a distributed program correctly implements a TLA+ [22] model in the sense that all *safety* properties enjoyed by the TLA+ model are also enjoyed by the program.

1.3 Overcoming Challenges in Fairneris

The approach taken by Fairneris to liveness reasoning is similar to Fairis: refinement-based reasoning. However, in order to abstract the details of the network from our liveness reasoning, we take special care of reflecting the network in the high-level model of Fairneris. The overall idea is that to prove liveness of a program e, the user of Fairneris, picks a model $\mathcal{U}$ and a relation ξ_U. The user then uses the program logic of Fairneris to show that e refines $\mathcal{U}$ under the relation ξ_U, which in turn implies that for any trace *ex* of the program e, there is a corresponding trace *utr* of $\mathcal{U}$, written $ex \preccurlyeq_{\xi_U} utr$, such that liveness properties that hold for *utr* also hold for *ex*. Thus, Fairneris allows analyzing the model $\mathcal{U}$ to prove that it satisfies the desired liveness property instead of the much more complicated distributed program e. We emphasize that the model $\mathcal{U}$ only reflects high-level network operation, *i.e.*, it is completely oblivious to messages being dropped and retransmitted; details will be explained later. To achieve this, we annotate the operational semantics of AnerisLang, the OCaml-like programming language of Aneris, with so-called *actions*, and construct a fixed network model $\mathcal{N}et$ and relation ξ_N relating the state of the network in the operational semantics to that of $\mathcal{N}et$ — constructed once for all AnerisLang programs. The model $\mathcal{N}et$ captures all the details of messages dropped, duplicated, *etc.* We then combine the model $\mathcal{U}$ with $\mathcal{N}et$, which we write as $\mathcal{U} \ltimes \mathcal{N}et$, in such a way that all sends and receives of $\mathcal{U}$ are synchronized to match the corresponding events in $\mathcal{N}et$. The program logic of Fairneris, behind the scene and unbeknownst to the user who is only dealing with $\mathcal{U}$, constructs a refinement between the program e, and the model $\mathcal{U} \ltimes \mathcal{N}et$ under the relation $\xi_U \ltimes \xi_N$. Crucially, for any trace utr of $\mathcal{U} \ltimes \mathcal{N}et$ we have that if $ex \preccurlyeq_{\xi_U \ltimes \xi_N} utr$, then $ex \preccurlyeq_{\xi_U} \pi(utr)$, where $\pi(utr)$ is a trace of $\mathcal{U}$ obtained from utr by projecting out the parts corresponding to $\mathcal{U}$.[1]

We note that it is far from obvious that the construction explained above regarding constructing a network model $\mathcal{N}et$ is conducive to a usable, modular program logic suitable for high-level reasoning. In other words, we need to answer the following question: can we obtain high-level reasoning principles, in terms of Hoare-logic rules, that allow us to reason directly about $\mathcal{N}et$ when we are in fact constructing a refinement relation relating the program to the model $\mathcal{U} \ltimes \mathcal{N}et$? To answer this question in the affirmative requires changes to the program logic of Fairis, notably introduction of a new *model update* logical connective.

1.4 Case Study

To demonstrate the applicability of our technique, we prove liveness and safety of the Stenning protocol [33], an algorithm to *reliably* transfer a stream of data between two nodes in an unreliable network. The Stenning protocol suitably exercises the challenges pertaining to safety and liveness properties in distributed systems, regarding retransmissions and distributed synchronization using sequence ids, and has thus previously been a target for verification. Ours is however the first work proving liveness of a low-level implementation of the Stenning protocol; see §6 for more details.

The examples and the case study of the Stenning protocol presented in this paper focus on addressing the core challenges as described above in §1.2. In particular, these examples do not feature node-local concurrency, which is readily taken care of in both Fairis and Aneris: we believe doing so would distract from the new ideas put forth in this paper.

1.5 Contributions

In this paper we make the following contributions:

(1) The Fairneris program logic for verifying trace refinement of distributed systems (§3)
(2) Verification of liveness of a low-level implementation of the Stenning protocol (§4)
(3) A labelled transition system for low-level UDP-based operational semantics (§5.2)
(4) A principle of model abstraction applied to network trace properties (§5.3)
(5) Fairneris as a shallow embedding on top of Trillium (§5.4)

All but one result of the paper have been mechanized in the Coq proof assistant. The only exception is the liveness theorem of the Stenning model, for which we instead give a paper proof.

We start by giving an overview of the Fairneris approach to verifying liveness properties of distributed systems (§2).

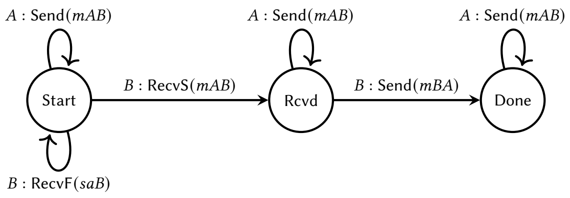
2 Overview

This section gives an overview of how Fairneris is used, and demonstrates how the main contributions of this paper enable proving liveness properties of distributed programs using a simple introductory example. The code for the two nodes, nodes A and B, of our example is given on

[1]In fact the model $\mathcal{U} \ltimes \mathcal{N}et$ once more to the model $\mathsf{Fuel}(\mathcal{U} \ltimes \mathcal{N}et)$ which hides details of the thread scheduler. However, the so-called fuel construction, $\mathsf{Fuel}(\cdot)$, is not new in our work; it is taken verbatim from Fairis.

```
nodeA saA saB =
  let sock = new_socket saA in
  while true do
    send sock saB "ping"

nodeB saB saA saB =
  let sock = new_socket saB in
  while recv sock = None do skip;
  send sock saA "done!"
```



$$mAB \triangleq (saA, \text{"ping"}, saB)$$
$$mBA \triangleq (saB, \text{"done!"}, saA)$$

Fig. 1. Retransmit example consisting of the program Σ_{retr} (left), and the model $\mathcal{U}_{retr}$ (right)

the left-hand-side column of Figure 1. We will write Σ_{retr} for this distributed system. The liveness property that we wish to show about Σ_{retr} is that node B eventually sends the message "done!" to node A. In Σ_{retr} node A repeatedly sends message "ping" to node B. Node B, on the other hand, repeatedly calls **recv** until it receives "ping" from A. And when it does, it sends back a message "done!", and terminates. Hence, *under fairness assumptions*, node B must eventually send "done!" to node A.[2]

Recall that, as we explained in §1, in Fairneris we prove liveness properties by establishing a refinement between the distributed program and a model $\mathcal{U}$ picked by the user of Fairneris. For our simple example the model $\mathcal{U}$ is the model $\mathcal{U}_{retr}$ given in the right-hand-side column Figure 1.

Below, we will describe how we state fairness and liveness as trace properties both for program traces and model traces, and discuss how we use these notions to prove that fair traces of the program are live (in this case: that the node B sends the message "done!" to node A). We first define fairness and liveness for the program execution in §2.1, and then show how this reasoning can be lifted to the level of $\mathcal{U}_{retr}$ in §2.2. Finally, we briefly discuss our methodology of using Fairneris's program logic to prove that the program refines $\mathcal{U}_{retr}$ in §2.3.

2.1 Fairness and Liveness in the Program

All throughout this paper, we will use *linear temporal logic* (LTL) formulas to state properties about potentially infinite executions of distributed systems or traces of abstract models. In a few words, the LTL operators we will use are "eventually", written $\mathbb{F}P$, which states that the property P must become true at a point of the execution; and "always", written $\mathbb{G}P$, which states that P holds at every step of the execution. We describe events in terms of the actions Send, RecvS, and RecvF, for sending, successfully receiving, or failing to receive, respectively. The Send and RecvS actions are indexed by message triples, consisting of the source address, contents, and destination address of the message. We can thus denote the sending of the message "done!" from B to A as Send(mBA), where $mBA \triangleq (saB, \text{"done!"}, saA)$. The RecvF action is indexed by the receiving socket address, e.g. RecvF(saB), indicating that the program attempted to retrieve a message from the buffer of the socket assigned to address saB but failed as the buffer was empty. The liveness property of interest can then be stated as follows:

$$\text{For all executions } ex \text{ of } \Sigma_{retr}, \quad ex \vDash \mathbb{F}\ \text{Send(mBA)} \qquad \text{(Live*)}$$

[2]Note that it is impossible to have two nodes simultaneously reach agreement. This phenomenon known as the two-general problem [2, appendix] occurs even when the network is completely reliable, as soon as there is any delay in communication. Hence, here we only show that one node, node B reaches the end of communication.

As we explained in §1, messages in Aneris and Fairneris can be non-deterministically dropped. Thus, even when a program, like node A in Σ_{retr}, sends the same message in an infinite loop, it is not guaranteed that it will be delivered. That is, the following is a valid trace of network events

$$\mathsf{Send}(mAB) \cdot \mathsf{Drop}(mAB) \cdot \mathsf{RecvF}\ (saB) \cdot \mathsf{Send}\ (mAB) \cdot \mathsf{Drop}\ (mAB) \cdot \mathsf{RecvF}\ (saB) \cdots$$

where each time the node A sends a message, it is immediately dropped by the network, preventing node B from ever receiving it. Such an execution violates our liveness property: $\mathbb{F}$ Send(mBA). Therefore, we consider following, usual, notion of network fairness expressed as an LTL formula:

$$\mathbb{G}(\mathbb{GF}\ \mathsf{Send}(msg) \implies \mathbb{F}\ \mathsf{Deliver}(msg)) \qquad \text{(NetFair)}$$

which states that, for all messages, it is always the case that, if a message is sent infinitely often (always eventually), then it is eventually delivered. Given (NetFair), the distributed system Σ_{retr} seems to satisfy the property (Live*): since node A sends its message infinitely often, according to (NetFair), the message is delivered at some point. Node B calls receive in a loop, so that it eventually retrieves the message from the buffer where it was previously delivered, and then it immediately sends the "done!" message.

This reasoning, and indeed the fact that Σ_{retr} satisfies (Live*), relies on the fact that both main threads of nodes A and B run “enough” times. (Fairneris's operational semantics, like Aneris's, consider one large thread pool consisting of all threads of all nodes.) We express fairness of the thread scheduler as the following LTL property

$$\mathbb{G}(\mathsf{thread\_enabled}(\zeta) \implies \mathbb{F}\ (\mathsf{thread\_steps}(\zeta) \lor \neg\mathsf{thread\_enabled}(\zeta))) \qquad \text{(SchedFair)}$$

where the predicate thread_enabled(ζ) means that the thread ζ is not finished, and is ready to take an execution step, and thread_steps(ζ) expresses that ζ does take an execution step. In other words, if a thread can run, then it eventually runs.

We are now able to state the correct result about Σ_{retr}, which makes explicit the fairness hypotheses we require about the scheduler and the network:

$$\text{For all executions } ex \text{ of } \Sigma_{retr}, \quad ex \vDash (\mathsf{SchedFair} \land \mathsf{NetFair}) \Rightarrow \mathbb{F}\ \mathsf{Send(mBA)} \qquad \text{(Live)}$$

This is the statement that we eventually prove about the system Σ_{retr} using our program logic, but before that, we will lift it to $\mathcal{U}_{retr}$ in §2.2.

Remark 2.1 (Locallity of Liveness). The property (Live) above is local in the sense that it is trace-wise, *i.e.*, it states that for any trace *ex*, if *ex* is fair, then eventually Send(mBA). This is key in Fairis's approach, and thus in that of Fairneris. It means that the program logic need not be concerned with fairness — the program logic neither enforces, nor relies upon fairness conditions. The program logic allows us to establish a refinement relation between programs and models, including both fair and unfair executions of the program. As we will discuss below, care must only be taken to ensure that fair traces of the program correspond to fair traces of the model.

Remark 2.2 (Fairness and Delivery in the Presence of Acknowdgements). The formulation of (NetFair) above is subtle: it is in fact not necessary to send the message infinitely often for the entire execution of the program to be sure that it is delivered. As we will see in the case study in §4, where a message, once acknowledged, is no longer retransmitted. That is, it suffices to only retransmit until the acknowledgement is received to guarantee that under (NetFair), all messages are eventually delivered, or indeed even stronger that all messages have their acknowledgements received by the sender. This is despite the fact that in all such fair executions of the system messages are only ever sent finitely many times, which might appear strange at first. To understand why this is the case, note that if the acknowledgement of a message is not eventually received, that message is sent infinitely often. Thus, by (NetFair) it must also be delivered infinitely often, and hence, also received infinitely often. But then, since the recipient receives the message infinitely often, it sends acknowledgements back infinitely often, which again by (NetFair) implies that the acknowledgement is eventually delivered and hence received. This contradicts our assumption that the acknowledgement is not received, and so we can conclude that progress is eventually made.

2.2 Fairness and Liveness in the Abstract Model $\mathcal{U}_{retr}$

Here, following the methodology of Fairis we simplify the reasoning about the program by instead reasoning about the model $\mathcal{U}_{retr}$. The idea, which we will make formal in §3, is that whenever we have $ex \preccurlyeq_{\xi_{retr}} utr$ for a trace ex of the program and a trace utr of $\mathcal{U}_{retr}$, then (Live) holds whenever (ModLive) below holds.

$$\text{For all traces } utr \text{ of } \mathcal{U}_{retr}, \quad utr \vDash \big(\text{ModSchedFair} \wedge \text{ModNetFair}\big) \implies \mathbb{F}\,\text{Send}(\text{mBA}) \qquad (\text{ModLive})$$

Here, ModSchedFair and ModNetFair are respectively model counterparts to SchedFair and NetFair above. Below, we will give a brief discussion of the model $\mathcal{U}_{retr}$, present ModSchedFair and ModNetFair, followed by arguments as to why (ModLive) holds.

The model $\mathcal{U}_{retr}$ has three states, corresponding to the three stages of progress of the distributed system Σ_{retr}: Start, the initial stage where node A is sending its message, Rcvd, where node B has received at least one of the messages from node A but has not yet replied, and, finally, Done where node B has sent the message "done!" to node A. Note how the model $\mathcal{U}_{retr}$ is significantly simpler than that of Σ_{retr}, considering operational semantics as an LTS. In particular, due to the non-deterministic behavior of the network dropping and duplicating messages, the loop in node A repeatedly sending a message, *etc.*, the state space of the Σ_{retr} is not even finite — this is also the case for the model $\mathcal{U}_{retr} \ltimes \mathit{Net}$ that we are not dealing with in this section. By contrast, $\mathcal{U}_{retr}$ has only three states.

The transitions of $\mathcal{U}_{retr}$ denote how actions from each node update the current stage of the distributed system. They are labeled with two pieces of information, separated by a colon: the role, to the left of the colon, and the action, to the right of the colon. For instance, the label $B : \text{RecvS}(mAB)$ states that the transition corresponds to the *action* of receiving the message from A (with socket address saA) to B (with socket address saB), with the contents "ping" (denoted as the triple $mAB \triangleq (saA, \text{"ping"}, saB)$), and that the *role* performing this transition be B. Roles, a concept we adopt from Fairis [34], are the logical counterpart to thread identifiers, denoting *who* is performing the action. This allows us to state scheduler fairness for models using a definition similar to (SchedFair), as follows:

$$\mathbb{G}\big(\text{role\_enabled}(\rho) \implies \mathbb{F}\,(\text{role\_steps}(\rho) \vee \neg\text{role\_enabled}(\rho))\big) \qquad (\text{ModSchedFair})$$

where ρ, universally quantified ranges over all roles in $\mathcal{U}_{retr}$, *i.e.*, either A or B, which correspond to the main threads of the two nodes A and B, respectively.

The notion of actions is a contribution of this work. Actions allow us to tie the network to the abstract model in a lightweight fashion. In particular, while the model mentions messages being sent or received, which reflect actions performed by the program itself, the abstract model $\mathcal{U}_{retr}$ is oblivious to the actions performed by the network itself, such as delivering or dropping a message. The absence of delivery actions (and hence also receive buffers) from $\mathcal{U}_{retr}$ is crucial for managing the complexity of formal reasoning about the network and its fairness. Had delivery actions been part of $\mathcal{U}_{retr}$, we would have had to track the position of messages of interest in the buffer whenever we reasoned about the network. On the other hand, the lack of delivery actions in the $\mathcal{U}_{retr}$ necessitates changing the notion of network fairness compared to (NetFair). The intuition is simple: if a sender (re)transmits messages infinitely often (or as argued earlier until acknowledgement), and the recipient repeatedly calls **recv**, then the recipient must eventually receive the message. This is stated formally as an LTL formula as follows:

$$\mathbb{G}(\mathbb{GF}\ \mathsf{Send}(msg) \quad\Rightarrow\quad \mathbb{GF}\ \mathsf{Recv\_from}(msg.dest) \quad\Rightarrow\quad \mathbb{F}\ \mathsf{RecvS}(msg)) \qquad (\mathsf{ModNetFair})$$

where Recv_from is a shorthand describing the action of calling **recv** on a socket bound to the destination address of the message *msg*, regardless of whether it succeeds or fails.

Now that we have defined both (ModSchedFair) and (ModNetFair), we proceed to argue why (ModLive) holds. As per scheduling fairness, it is clear that it is sufficient to reach the Rcvd state, as we cannot repeatedly take the transition A : Send(mAB) in that state, and hence must eventually take the transition by the role B which sends the "done!" message. In turn, proving that Rcvd is reached amounts to proving that the transition labeled with RecvS(mAB) is eventually taken. This follows directly from (ModNetFair). Assume that this transition is never taken. In that case, we stay in the state Start forever. However, staying in Start forever, by fairness of scheduling, satisfies the two premises of (ModNetFair). And hence, the transition RecvS(mAB) must eventually be taken, which contradicts our assumption.

2.3 A Program Logic for Refinement

In this section we present the basic, high-level ideas underlying how the program logic of Fairneris can be used to conclude that the program Σ_{retr} refines the model $\mathcal{U}_{retr}$. Here, we will only discuss the specifications for the distributed program Σ_{retr} in terms of Hoare triples, and discuss the intuition and high-level ideas of the proof. We leave the formal, more detailed proof, which we will present in §3.3, for after we have presented the program logic and its inference rules in more details. Note that the specifications presented in this section, apart from including actions, are virtually identical to what one would write in Fairis. Our contribution, in terms of the program logic of Fairneris, is the novel proof rules and how they are proven sound which we will present and discuss in §3 and §5.4 respectively.

As we will discuss in more detail in §3.3, the crux of the proof that Σ_{retr} refines $\mathcal{U}_{retr}$ is to show the following two Hoare triples (defined in terms of weakest preconditions in the usual way):

$$\{I_{retr} * \zeta_A \Mapsto \{A := 42\} * \ldots\}^{\zeta_A}\ \mathsf{nodeA}\ \mathsf{saA}\ \mathsf{saB}\ \{\zeta_A \Mapsto \emptyset\}$$

$$\{I_{retr} * \zeta_B \Mapsto \{B := 42\} * \circ_{\gamma_B}(\mathsf{Start}) * \ldots\}^{\zeta_B}\ \mathsf{nodeB}\ \mathsf{saA}\ \mathsf{saB}\ \{\zeta_B \Mapsto \emptyset\}$$

where the invariant I_{retr} is as follows (in Iris's notation we write $\boxed{P}$ to say that P must always hold throughout the proof):

$$I_{retr} \triangleq \boxed{\exists m,\ \circ_{\gamma_M}(m)\ *\ \bullet_{\gamma_B}(m)}$$

We omit some details for now (denoted ...), but will revisit the full specification and proof in §3.3.

Let us unpack and explain these specifications. The logical connective $*$ is the separating conjunction, the special resource-aware conjunction of separation logic. The logical statement $P * Q$ states that both P and Q hold, but resources they assert ownership over are disjoint. The predicate $\zeta \Mapsto \{\rho := f\}$, as in Fairis, indicates the locale ζ is associated to the role ρ and that this role has *fuel* f. This is to be understood as both a permission, and an obligation to perform ρ actions. It is a permission in the sense that a thread can only perform a ρ action if it is assigned that role. On the other hand, in Fairneris just as in Fairis, when a thread is assigned a role ρ, it can only postpone performing a ρ-action for f steps, as each stutter step decreases its fuel. The only way to increase it is to perform a ρ-action. This restriction of finite postponement, as we will explain in §5.3, is key in ensuring that (SchedFair) implies (ModSchedFair). In this sense, the main thread of node A having role A assigned to it, *must* infinitely often perform a send operation as described by $\mathcal{U}_{retr}$, because in $\mathcal{U}_{retr}$ all states have such send loops. Similarly, the postcondition $\zeta \Mapsto \emptyset$ indicates the thread may only terminate if it has fulfilled all its obligations. The main thread of node A satisfies this condition because it never terminates. The main thread of node B on the other hand only terminates when we are in state Done which has no B-transitions, and hence no B-obligations. The invariant I_{retr} states that we are, at all times in some state m of $\mathcal{U}_{retr}$, indicated by the resource $\circ_{\gamma_{\mathcal{M}}}(m)$ in I_{retr}. The resource $\bullet_{\gamma_B}(m)$ in I_{retr} and $\circ_{\gamma_B}(\text{Start})$ in the precondition above, as we will explain, allow the thread ζ_B to keep track of the current state of $\mathcal{U}_{retr}$, even though it is shared with thread ζ_A via the invariant. The construction of $\bullet_{\gamma_B}(m)$ and $\circ_{\gamma_B}(m)$ in terms of Iris resources, the so-called agreement resource algebra [19], ensures that these two predicates always agree on the state they track. This is codified in the logic as the following two rules:

own-agree
$$\bullet_{\gamma_B}(m) * \circ_{\gamma_B}(m') \vdash m = m'$$

own-update
$$\bullet_{\gamma_B}(m) * \circ_{\gamma_B}(m) \vdash \Rrightarrow \bullet_{\gamma_B}(m') * \circ_{\gamma_B}(m')$$

where $\Rrightarrow$ is the so-called update modality, $\Rrightarrow P$ holds whenever P holds after updating resources, which is allowed throughout the proof.

For node A, we see that it simply performs a send operation in a loop; hence, each send follows the previous one after finitely many steps. Thus, we only need to show that each send operation performs an A-action. For this we need to know that we have the permission/obligation to do so. This is reflected in the fact that we are assigned role A. By the invariant I_{retr} we know that we are in some state m, indicated by $\circ_{\gamma_{\mathcal{M}}}(m)$. Since all states have an $A : \text{Send}(mAB)$ transition, we always can/should perform the send operation. Furthermore, all these transitions are loops and hence never change the state tracked by $\circ_{\gamma_{\mathcal{M}}}(m)$, always maintaining the invariant.

For node B, initially, we know that we are in state Start, because we have $\circ_{\gamma_B}(\text{Start})$, and by own-agree, the m in the invariant I_{retr} must be Start. In this state, taking the loop $B : \text{RecvF}(saB)$ is completely analogous to taking $A : \text{Send}(mAB)$, we discussed above. However, when the Recv operation succeeds, we go from Start to Rcvd in $\mathcal{U}_{retr}$. That is, after taking that step we will have $\circ_{\gamma_{\mathcal{M}}}(\text{Rcvd})$. At this point, we will update resources using the rule own-update to obtain $\bullet_{\gamma_B}(\text{Rcvd})$ and $\circ_{\gamma_B}(\text{Rcvd})$ which maintains the invariant I_{retr}. Finally, in state Rcvd we have only one permission/obligation, *i.e.*, to take the transition $B : \text{Send}(mBA)$. We perform this action, and update our state to $\bullet_{\gamma_B}(\text{Done})$ and $\circ_{\gamma_B}(\text{Done})$. In this final state there are no B-obligations left and hence the thread *must* terminate, because there is no action left in its assigned role.

3 Fairneris Program Logic

The primary goal of Fairneris is to verify that a $\preccurlyeq_{\xi_U}$ refinement holds between a program and a given user model. This includes the user picked refinement relations ξ_U that must hold between the program and model states for any step of the program.

The primary benefit of the Fairneris logic is that it *internalises* the parts of the global model that does not relate to the user model. That is, it hides the irrelevant model aspects of the fuel and network model instrumentations, and relegate these (still crucial) parts of the verification to the program logic. We hide the fuel instrumentation by leveraging pre-existing ideas from Fairis [34]. To interface with the network instrumentation—updating the user model *non-deterministically* based on the action that it emits—we introduce a novel so-called "model update" construction, alongside a pre-existing approach to reasoning about distributed systems from Aneris [20].

In this section we present the Fairneris program logic. We first present the adequacy theorem (§3.1), defined entirely in terms of the user model. We then present the rules of Fairneris (§3.2), which are used to prove the weakest precondition and that the user-picked refinement relation holds invariably. Finally, we remark on how the Fairneris logic is used to verify the refinement of the retransmit example from Figure 1 (§3.3).

3.1 Fairneris Adequacy

The Fairneris adequacy theorem formalises that the refinement $\preccurlyeq_{\xi_U}$ holds, if we can prove that (1) weakest preconditions for all initial threads: $\mathrm{wp}^{\zeta}_{\mathcal{E}}\ e\ \{\Phi\}$; and (2) the user-picked refinement relation ξ_U always holds: $\mathit{AlwaysHolds}(\xi_U)$, under the assumption of initial separation logic resources for (i) the model state (*à la* Trillium); (ii) the fuel instrumentation (*à la* Fairis); (iii) the network state (*à la* Aneris); and (iv) initially allocated invariants.

The weakest precondition $\mathrm{wp}^{\zeta}_{\mathcal{E}}\ e\ \{\Phi\}$ denotes that the expression e is safe to execute, and if it terminates with value w, then $\Phi\ w$ holds. The weakest precondition is instrumented with its locale $\zeta \in \mathit{Ip} \times \mathbb{N}$, denoting the ip address of the node in the distributed system that the expression belongs to, along with its thread id on that node. Finally, the weakest precondition has a "mask" $\mathcal{E}$, that denotes which invariants it can open. We sidestep the technical details of masks for now.

The proof of the weakest precondition and the user-picked refinement relation can rely on initially consistent resources that respectively assert properties about the model state, the fuel instrumentation, and the network. Based on these initial resources, we can additionally allocate invariants that are then guaranteed to hold throughout the entire execution of the program (via the weakest preconditions) *inside* of the separation logic. These initially allocated invariants are imperative, as they allow us to prove the user-picked refinement relations between the program and model states. Formally, this works out as we can access any properties stored in invariants, when proving the desired user-picked refinement relations $\mathit{AlwaysHolds}(\xi_U)$.

The formal statement of the adequacy theorem is as follows:

Theorem 3.1 (Fairneris-Adequacy). *Let* tp *be a node-threadpool* $((\mathit{Ip} * \mathbb{N}) \rightharpoonup \mathit{Expr})$ *of the initial program threads. Let* $\mathsf{m} \in \mathcal{U}$ *be a model state, and* ξ_U *be a relation between program and model states. Let* F_{init} *be an initial fuel map* $((\mathit{Ip} * \mathbb{N}) \to (\mathbf{role} \to \mathbb{N}))$ *with all* $\zeta \in \mathrm{dom}(\mathsf{tp})$ *assigned to respective role maps* fs *mapping the roles of* m *to a user picked concrete fuel limit* F_{cap}*. All roles of* m *are uniquely covered in the initial fuel map. If*

$$\begin{aligned}&\big(\circ_{\gamma_{\mathcal{M}}}(\mathsf{m}) * (\textstyle\mathop{\ast}_{\zeta \mapsto fs \in \mathsf{F}_{init}} \zeta \Rrightarrow fs) \ * \mathsf{NetRes}(A)\big) \mathrel{-\!\!*} \\ &\tilde{\Rrightarrow}_{\top} \mathit{AlwaysHolds}(\xi_U) * \mathop{\Large\ast}_{\zeta \mapsto e \in \mathsf{tp}} wp^{\zeta}_{\top}\ e\ \{w.\ \zeta \Rrightarrow \emptyset\}\big)\end{aligned}$$

then for all fair executions ex of c there exists a fair trace utr of m *where ex* $\preccurlyeq_{\xi_U}$ *utr holds in the meta-logic.*

Here $\circ_{\gamma_{\mathcal{M}}}(\mathsf{m})$ reflects the state of the user model, $(\mathop{\ast}_{\zeta \mapsto fs \in \mathsf{F}_{init}} \zeta \Rrightarrow fs)$ the state of the fuel instrumentation of Fairis, and $\mathsf{NetRes}(A)$ is the initial network resources of Aneris. The modality $\tilde{\Rrightarrow}_{\top}$ allow us to allocate invariants, which are then enforced by the proofs of the individual WPs, and thereby available when proving $\mathit{AlwaysHolds}(\xi_U)$. We cover each of these constructs in more detail in the following section.

We remark that in the original Trillium [34] adequacy theorem, ξ_U is a *history-sensitive* refinement, indexed by the full traces of the program and model. This is also the case in Fairneris although we do not cover that in the adequacy theorem here for brevity sake.

3.2 Verification in Fairneris

The crux of applying the Fairneris logic is to verify the global refinement relation $\mathit{AlwaysHolds}(\xi_U)$, and the weakest preconditions of Fairneris. In this section we first cover how the former is achieved via the invariants of Fairneris and then discuss how the latter is achieved via the Fairneris rules for the weakest preconditions.

Fuel & model rules (outer program logic)

WP-STEP
$$\frac{{}^{\mathcal{E}}\tilde{\Rrightarrow}^{\mathcal{E}'}\, \mathsf{wp}^{\zeta}_{\mathcal{E}'}\, e \left\langle e'; \alpha^{?}.\ \mathsf{mu}^{\zeta;\alpha^{?}}_{\mathcal{E}'}\ {}^{\mathcal{E}'}\tilde{\Rrightarrow}^{\mathcal{E}}\, \mathsf{wp}^{\zeta}_{\mathcal{E}}\, e'\ \{\Phi\} \right\rangle}{\mathsf{wp}^{\zeta}_{\mathcal{E}}\, e\ \{\Phi\}}$$

WP-ROLE-DEALLOC
$$\frac{\begin{array}{c}\circ_{\gamma_{\mathcal{M}}}(\mathsf{m}) \qquad \mathsf{m} \overset{\rho\,:}{\not\longrightarrow} \_ \qquad \zeta \Mapsto \{\rho := \_\} \uplus fs \\ (\circ_{\gamma_{\mathcal{M}}}(\mathsf{m}) * \zeta \Mapsto fs) \mathrel{-\!\!*} \mathsf{wp}^{\zeta}_{\mathcal{E}}\, e\ \{\Phi\}\end{array}}{\mathsf{wp}^{\zeta}_{\mathcal{E}}\, e\ \{\Phi\}}$$

MDL-STEP-FUEL
$$\frac{\begin{array}{c}\zeta \Mapsto fs^{++} \qquad fs \neq \emptyset \\ (\zeta \Mapsto fs \mathrel{-\!\!*} P)\end{array}}{\mathsf{mu}^{\zeta}_{\mathcal{E}}\ P}$$

MDL-STEP-MODEL
$$\frac{\begin{array}{c}\circ_{\gamma_{\mathcal{M}}}(\mathsf{m}) \qquad \mathsf{m} \xrightarrow{\rho\,:\,\alpha^{?}} \mathsf{m}' \\ \zeta \Mapsto \{\rho := \_\} \uplus (fs^{++}) \\ ((\circ_{\gamma_{\mathcal{M}}}(\mathsf{m}') * \zeta \Mapsto \{\rho := \mathsf{F}_{cap}\} \uplus fs) \mathrel{-\!\!*} P)\end{array}}{\mathsf{mu}^{\zeta;\alpha^{?}}_{\mathcal{E}}\ P}$$

SOCKET-INTERP-ALLOC
$$\frac{\mathsf{Unallocated}(\{sa\})}{\tilde{\Rrightarrow} sa \Mapsto \Phi}$$

Program rules (inner program logic; an excerpt)

SSWP-CREATE-SOCKET
$$\frac{\zeta.ip = sa.ip \qquad \mathsf{Unbound}(sa)}{\mathsf{wp}^{\zeta}_{\mathcal{E}}\ \texttt{new\_socket}\ sa\ \langle w.\, \exists sh.\, w = sh * sh \hookrightarrow sa \rangle}$$

SSWP-SEND
$$\frac{\begin{array}{c}\zeta.ip = msg.\mathsf{src.ip} \qquad sh \hookrightarrow msg.\mathsf{src} \\ msg.\mathsf{dst} \Mapsto \Phi \qquad msg.\mathsf{src} \rightsquigarrow (R, T) \qquad (msg \notin T \Rightarrow \Phi\ msg)\end{array}}{\mathsf{wp}^{\zeta}_{\mathcal{E}}\ \texttt{send}\ sh\ msg.\mathsf{str}\ msg.\mathsf{dst} \left\langle w; \alpha.\ \begin{array}{l} w = |msg.\mathsf{str}| * \alpha = \mathsf{Send}(msg)\ * \\ msg.\mathsf{src} \rightsquigarrow (R, T \cup \{msg\}) * sh \hookrightarrow msg.\mathsf{src}\end{array} \right\rangle}$$

SSWP-RECV
$$\frac{\zeta.ip = sa.\mathsf{ip} \qquad sh \hookrightarrow sa \qquad sa \rightsquigarrow (R, T) \qquad sa \Mapsto \Phi}{\begin{array}{c}\mathsf{wp}^{\zeta}_{\mathcal{E}}\ \texttt{recv}\ sh \\ \left\langle w; \alpha.\ \begin{array}{c}(w = \texttt{None} * \alpha = \mathsf{RecvF}(sa) * sh \hookrightarrow sa * sa \rightsquigarrow (R, T)) \vee \\ (\exists msg.\ w = \texttt{Some}\,(msg.\mathsf{str}, msg.\mathsf{src}) * \alpha = \mathsf{RecvS}(msg) * msg.\mathsf{dst} = sa\ * \\ sh \hookrightarrow sa * sa \rightsquigarrow (R \cup \{msg\}, T) * (msg \notin R \Rightarrow \Phi\ msg))\end{array} \right\rangle\end{array}}$$

Fig. 2. The Fairneris Program Logic

Invariants and User Relation Refinement Properties. Fairneris like Trillium and Iris before it makes use of separation logic invariants to determine global properties $\boxed{P}^{\mathcal{N}}$ that hold in between any step of execution. In separation logic invariants are primarily used for safely sharing otherwise exclusive resources between threads. In Trillium, and thereby Fairneris we can additionally leverage the global properties enforced by the invariants to verify global (safety) properties about the user-picked refinement relation ξ_U. In particular, the user has to prove $\tilde{\Rrightarrow}_{\top} \mathit{AlwaysHolds}(\xi_U)$, which allows opening all invariants, and discerning the state of the program and model via the available separation logic resources, such as the local view of the user model state $\circ_{\gamma_{\mathcal{M}}}(m)$.

Fairneris Weakest Precondition Rules. The weakest precondition of Fairneris captures that a program is safe to execute, and that the program refines the user model. Following the principles of Trillium, the role of the program logic is to prove an initial lockstep refinement; that the program is in one-to-one correspondence with the model. The lockstep refinement is further refined into the $\preccurlyeq_{\xi_U}$ refinement yielded by the adequacy theorem, after the fact, as will be explained in §5.3.

The proof of the weakest precondition mimicks the lockstep structure. At every step of the program, we must (1) prove that the program step is safe to execute, and (2) that we can update the model correspondingly. To achieve this reasoning principle we employ two constructions:

- The single-step weakest precondition (SSWP): $\mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \langle \Phi \rangle$
- The model update construction (MU): $\mathsf{mu}^{\zeta;\alpha^?}_{\mathcal{E}}\ P$

The SSWP is inspired by the Fairis logic [34]. It captures that the expression e is safe to execute for *one* step, after which the predicate Φ holds for the resulting expression e'. In Fairneris it additionally optionally emits any exhibited action $\alpha^?$. The MU is novel, and captures that the model is currently *out-of-sync* with the program by one step. It is parametric in optional actions $\alpha^?$, which lets us update the model based on the *non-deterministic* results of the SSWP, which we leverage when resolving receives, which can either fail or succeed. The decomposition is imperative as it lets us define and prove rules for each construction orthogonally of each other, ultimately avoiding rule duplication. With the decomposition in place, we obtain the rules of Fairneris shown in Figure 2. We often use the following syntactic sugar:

$$\begin{aligned}
\mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \{w.Q\} &\triangleq \mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \{\lambda w.\ Q\} & \mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \langle w;\alpha^?.Q\rangle &\triangleq \mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \langle \lambda(w,\alpha^?).\ Q\rangle \\
\mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \{Q\} &\triangleq \mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \{\lambda w.\ w = () * Q\} & \mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \langle w.Q\rangle &\triangleq \mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \langle \lambda(w,\alpha^?).\ \alpha^? = \bot * Q\rangle \\
\mathsf{mu}^{\zeta}_{\mathcal{E}}\ P &\triangleq \mathsf{mu}^{\zeta;\bot}_{\mathcal{E}}\ P & \mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \langle Q\rangle &\triangleq \mathsf{wp}^{\zeta}_{\mathcal{E}}\ e\ \langle \lambda(w,\alpha^?).\ w = () * \alpha^? = \bot * Q\rangle
\end{aligned}$$

The rules of Fairneris combine and extend the prior reasoning principles of Fairis [34] and Aneris [20], with the action exhibiting behaviour. In particular, we borrow the ideas for handling fuel from Fairis, while borrowing the ideas for reasoning about networks from Aneris.

The crux of the decomposition is the WP-STEP rule. The rule states that to prove a WP, we can (1) prove a single step of the program via the SSWP, and (2) reestablish consistency with the model via the MU, and (3) prove the rest of the program via the residual WP. The rule permits opening invariants around the SSWP/MU step via the ${}^{\mathcal{E}}\overset{\sim}{\Rrightarrow}{}^{\mathcal{E}'}$ before the SSWP and the ${}^{\mathcal{E}'}\overset{\sim}{\Rrightarrow}{}^{\mathcal{E}}$ after the MU. This is crucial as the properties enforced by invariants might apply to both the safety proof (governed by SSWP) and the corresponding model update (governed by MU).

The SSWP rules are directly inspired by Aneris, and are based on Aneris resources, extracted from the initial network resources yielded by the adequacy theorem:

$$\mathsf{NetRes}(A) \triangleq \mathsf{Unallocated}(A) * \mathsf{Unbound}(A) * \mathop{\huge *}_{sa\in A} sa \rightsquigarrow (\emptyset,\emptyset)$$

Most notably, the SSWP-SEND and SSWP-RECV rules have been instrumented to emit their respective actions. The rules follow the Aneris structure, where resources Φ are only attached to the initial message that is sent/received. We refer the interested reader to prior papers on Aneris [14, 20].

The MU rules are directly inspired by Fairis. The MDL-STEP-FUEL allow resolving non-model (unlabeled) "stuttering" steps, by burning fuel for the respective locale ζ. The MDL-STEP-MODEL allow resolving model steps, where the model transition has to reflect the optional action of the MU. That is, if the MU has a send label, then the model must take a send transition. If the MU does not have a label, the model must take an unlabeled transition. This means any network transition of the program *must* be reflected by the *user model.* While it is theoretically possible to extend the system to support non-model network transitions, we deemed that the increased complexity was not worth it. Finally, the WP-ROLE-DEALLOC rule allow for deallocation of roles once they have no transitions (and thus have concluded their lifetime), which is used to reach the post-condition of terminating programs, such as node B of the retransmit example.

The $\tilde{\Rrightarrow}_{\mathcal{E}}$ modality is a variant of the $\Rrightarrow_{\mathcal{E}}$ update modality covered in §2.3 which additionally has access to the program state. It obeys all of the relevant rules of $\Rrightarrow_{\mathcal{E}}$, while also enabling the allocation of Aneris "socket interpretations" via the Socket-interp-alloc rule, which was previously only possible in the presence of weakest preconditions. Most notably, the modality allow us to allocate socket interpretations at the start of a proof, which can then be added to the initial invariants used in the adequacy theorem, or be shared by the individual weakest precondition subproofs.

3.3 Verification of Retransmit Example

With the Fairneris logic in hand, we can now verify the model refinement of the retransmit example. We first prove the weakest preconditions of the individual nodes:

$$\left\{\begin{matrix} I_{retr} * \zeta_A \Rrightarrow \{A := 42\} * \mathsf{Unbound}(sa_A) * \\ sa_B \Rrightarrow (\_ = mAB) * sa_A \rightsquigarrow (\emptyset, \emptyset) \end{matrix}\right\}_{\zeta_A} \text{nodeA saA saB} \{\zeta_A \Rrightarrow \emptyset\}$$

$$\left\{\begin{matrix} I_{retr} * \zeta_B \Rrightarrow \{B := 42\} * \circ_{\gamma_B}(\mathsf{Start}) * \mathsf{Unbound}(sa_B) * \\ sa_B \Rrightarrow (\_ = mAB) * sa_A \Rrightarrow (\_ = mBA) * sa_B \rightsquigarrow (\emptyset, \emptyset) \end{matrix}\right\}_{\zeta_B} \text{nodeB saA saB} \{\zeta_B \Rrightarrow \emptyset\}$$

The proofs follow almost immediately from symbolic execution using the decomposition rules and the SSWP rules of Fairneris. The auxiliary steps (such as the reduction of the while-loops) are simply resolved via burning fuel. The model steps are resolved by opening the invariant to obtain the model state resource, and then taking the corresponding transitions in the model. Finally, the while loops are resolved using Löb induction, as is standard in step-indexed separation logics.

With the proven weakest preconditions, we can simply use the Fairneris adequacy theorem. The theorem provides all of the necessary initial resources to satisfy the preconditions of the WP proofs. The retransmit example does not rely on any user-picked refinement relation (thus $\xi_U(\sigma, m) := \mathsf{True}$), and thereby $AlwaysHolds(\xi_U)$ is trivial.

We thus obtain the refinement $\preccurlyeq_{\xi_U}$ between the distributed system and the retransmit model show in Figure 1.

4 Case Study: Stenning Protocol

To demonstrate the expressive power of Fairneris, we implement the Stenning protocol [33], and we verify the eventual progress of that implementation. The Stenning protocol is a reliable transfer protocol, used to transmit a stream, in order, from one node (the sender) to another (the receiver). The protocol uses a sliding window of 1, meaning that a new message is only sent once the prior message is acknowledged. The sliding window is achieved using sequence and acknowledgement ids for the sender and receiver, respectively. To focus on the liveness part of the verification effort, we simplify the protocol by only transferring the sequence ids (with no payload), and to loop endlessly. The property that we want to prove is thus that for any sequence id i, the receiver eventually sends an acknowledgement.

In this section we first cover the (simplified) Stenning protocol, its implementation in AnerisLang, and the formal liveness property that we want to prove (§4.1). We then give a formal model of the protocol and the corresponding liveness property in the model, and provide a paper proof of the property (§4.2). We finally conclude the proof of the liveness property at the implementation level, through a refinement between the implementation and the model via Fairneris (§4.3).

4.1 Stenning Protocol and its Implementation

The goal of the sender is to send the sequence $0, 1, 2, \ldots$. Since the network is unreliable, it waits for an acknowledgment (ACK) for i before sending $i + 1$. In the sender side the situation is quite simple: as long as it does not receive the ACK for the current number —either because it received

```
stenningA saA saB =
  let shA = new_socket saA in
  socket_bind shA saA;
  (rec f i =
    send shA saB i;
    match (recv shA) with
    | None   => f i
    | Some m => if m.2 = saB
                then
                 let j = m.1 in
                  if i = j
                  then f (i+1)
                  else f i
                else
                  f i) 0
```

```
stenningB saB saA =
  let shB = new_socket saB in
  socket_bind shB saB;
  (rec f j =
    match (recv shB) with
    | None   => f j
    | Some m => if m.2 = saA
                then
                  let i = m.1 in
                  if j+1 = i
                  then send shB saA i; f i
                  else send shB saA j; f j
                else
                  f j) -1
```

$$mAB\ i \triangleq (saA, \mathrm{i}, saB) \qquad\qquad mBA\ j \triangleq (saB, \mathrm{j}, saA)$$

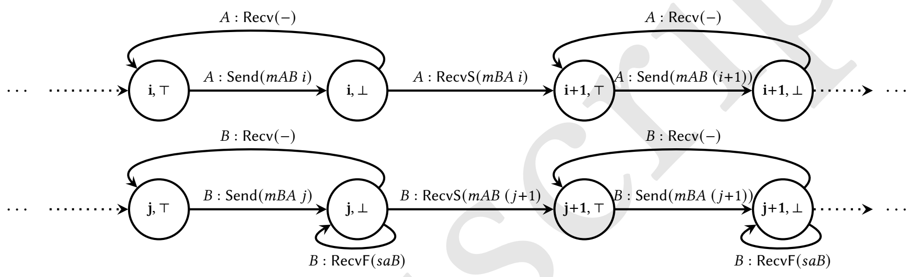


Fig. 3. The implementation and model of the Stenning protocol

one for the wrong id or because it received nothing after a timeout— it sends the current id again. If it receives the right ACK, it proceeds to send the next number. The receiver is almost dual to the sender: it repeatedly sends an ACK for the highest id j it has received yet. When it receives the next id, $j + 1$, it sends an ACK and continues. The subtlety is that it only resends an ACK when it receives an outdated message (whose id is $\leq j$). This way, the number of ACKs sent is much lower. The implementation of the Stenning protocol is shown in Figure 3.

Similar to the retransmission example, the Stenning protocol only makes progress when we consider fair scheduling and a fair network. Formally, the property that we want to prove is:

$$\text{For all executions } ex \text{ of } \Sigma_{sten},\ \ ex \vDash \big(\mathsf{SchedFair} \wedge \mathsf{NetFair}\big) \implies (\forall i.\, \mathbb{F}\ \mathsf{Send}(\mathsf{mBA\ i}))$$

4.2 Stenning Protocol Model, Liveness Property, and Proof

To formally model the Stenning protocol we model the sender and receiver individually, and construct the user model as the cartesian product of the two. The individual models can be found in Figure 3. We model the state of the sender as its current sequence id i, and whether it is in sending ($\top$) or receiving ($\bot$) state. Similarly, we model the state of the receiver as its current acknowledgement id j, and whether it is in sending ($\top$) or receiving ($\bot$) state. The initial state of the model is $((0, \top), (-1, \bot))$.

The model transitions directly correspond to the transitions of the protocol. For example, when the sender in state $(i, \bot)$ receives the expected acknowledgement i, it transitions to $(i + 1, \top)$. Conversely, when the receiver in state $(j, \bot)$ receives the expected sequence id $j+1$, it transitions to $(j + 1, \top)$. We use Recv($-$) to denote wildcard receive transitions matching any uncovered receive transition.

A caveat of this model is that it does not rule out invalid states, where the sender and receiver are out of sync; i.e. where the sequence id and acknowledgement id are more than one apart, this is because user models do not give any semantics to the send and receive labels they carry. Traces that reach these invalid states do not, in general, satisfy the liveness property we want, we therefore need to rule them out. To do so, we need an additional assumption

$$\text{For all trace } utr \text{ of } \mathcal{M}_{sten},\ \ utr \vDash \quad \mathbb{G}(i = j \lor i + 1 = j) \qquad \text{(At-Most-One-Off)}$$

This assumption needs not be trusted: we will prove *using the program logic* that any execution of the program in Figure 3 refines a trace of $\mathcal{U}_{sten}$ that satisfies (At-Most-One-Off). In summary, we prove the following property about the model $\mathcal{U}_{sten}$

Lemma 4.1. *For all trace utr of* $\mathcal{M}_{sten}$,

$$utr \ \vDash \ (\mathsf{ModSchedFair} \land \mathsf{ModNetFair} \land \textit{At-Most-One-Off}) \implies (\forall i.\, \mathbb{F}\ \mathsf{Send(mBA\ i)}).$$

With the assumptions and model in hand, proof of the liveness property can be carried out. We only sketch the first case, to show how the safety property is crucial in the proof.

Proof. Suppose the current state satisfies $i + 1 = j$, as for the initial state. Let us show that eventually j is incremented by exactly one. According to (At-Most-One-Off) and because there are no transitions that either decrease the values of i or j, or increase both of them, there are two cases: either j is incremented by one (and we conclude), or the values of i and j stay constant forever. In that case, it is easy to see that, according to (ModSchedFair), the message i is sent infinitely often, and B receives infinitely often. We can also conclude, as (ModNetFair) implies that the transition receiving i must be eventually be taken, increasing j. We therefore reach a state where $i = j$, and we can prove that i eventually increases by a similar, if slightly more involved, argument. □

4.3 Stenning Protocol Refinement in Fairneris

To transport the liveness property for the model to the implementation, we need to prove a refinement between them, with a user-picked refinement relation that lets us derive the (At-Most-One-Off) safety property for the model trace. We achieve both of these via the Fairneris program logic. In particular, we first pick a refinement relation that lets us derive the At-Most-One-Off property:

$$\xi_{sten}(\_, ((i, stA), (j, stB))) \triangleq (i = j \lor i + 1 = j)$$

Since the first argument is not used here, this amounts to the fact that $i = j \lor i = j - 1$ is an invariant of the model trace, which is precisely (At-Most-One-Off).

We then pick an invariant that subsumes this relational property:

$$I_{sten} \triangleq \boxed{\begin{array}{c}\exists i, stA, j, stB.\ \circ_{\gamma_M}((i, stA), (j, stB)) * \bullet_{\gamma_A}(stA) * \bullet_{\gamma_B}(stB) * \\ \bullet_{\gamma_i}(i) * \bullet_{\gamma_j}(j) * (i = j \lor i + 1 = j)\end{array}}$$

Similar to the retransmit example, the invariant owns the model state via $\circ_{\gamma_M}((i, stA), (j, stB))$, and exposes the individual parts of the state via the remaining ghost state. Finally, the invariant directly enforces the ξ_{sten} refinement relation. What remains to be done is to prove the weakest preconditions of the sender and receiver in the presence of the invariant, and to apply the Fairneris adequacy theorem.

The crux of verifying the Stenning implementation using this invariant is to appropriately delegate the local views of the state between the sender and receiver, so that they can adequately make progress, while preserving the synchronicity property. Notably, the sender and receiver need to deduce that any increment to their counter does not invalidate the invariant. They thus need a part of the view of the other participant, whenever they are about to make progress. This is achieved via so-called *fractional ownership*, which has the following rules:

OWN-FRAC-SPLIT
$$\circ_\gamma(m) \dashv\vdash \circ_\gamma^{1/2}(m) * \circ_\gamma^{1/2}(m)$$

OWN-FRAC-AGREE
$$\circ_\gamma^{1/2}(m) * \circ_\gamma^{1/2}(m') \vdash m = m'$$

That is, we can split the local view in two, and unify values between them. With this we can enforce that the sender and receiver "take turns" incrementing their respective part of the state, effectively enforcing the (At-Most-One-Off) property. In particular, we let the current node borrow half of the other node's local view in order to make progress. When it has incremented its counter, it "yields" the progress permission by sending its borrowed local view along with half of its own view when sending a fresh sequence/acknowledgement id. We can achieve this delegation of the resources via the Aneris socket interpretation, using the following protocols:

$$\begin{aligned}\Phi_A\ msg &\triangleq \circ_{\gamma_i}^{1/2}(msg.\mathrm{str}) * \circ_{\gamma_j}^{1/2}(msg.\mathrm{str})\\ \Phi_B\ msg &\triangleq \circ_{\gamma_i}^{1/2}(msg.\mathrm{str}) * \circ_{\gamma_j}^{1/2}(msg.\mathrm{str} - 1)\end{aligned}$$

Note that the fractional ownership prohibits one from updating the ownership in the invariant.

With the invariant and socket interpretations we only need to prove the following weakest preconditions (presented as Hoare triples):

$$\left\{\begin{array}{l} I_{sten} * sa_A \Mapsto \Phi_{\mathrm{A}} * sa_B \Mapsto \Phi_{\mathrm{B}} * sa_A \rightsquigarrow (\emptyset, \emptyset) * \mathsf{Unbound}(sa_A) * \\ \zeta_A \Mapsto \{A := 42\} * \circ_{\gamma_i}(0) * \circ_{\gamma_j}^{1/2}(-1) * \circ_{\gamma_A}(\top) \end{array}\right\}^{\zeta_A} \mathrm{stenningA\ saA\ saB}\ \{\zeta_A \Mapsto \emptyset\}$$

$$\left\{\begin{array}{l} I_{sten} * sa_A \Mapsto \Phi_A * sa_B \Mapsto \Phi_B * sa_B \rightsquigarrow (\emptyset, \emptyset) * \mathsf{Unbound}(sa_B) * \\ \zeta_B \Mapsto \{B := 42\} * \circ_{\gamma_j}^{1/2}(-1) * \circ_{\gamma_B}(\bot) \end{array}\right\}^{\zeta_B} \mathrm{stenningB\ saA\ saB}\ \{\zeta_B \Mapsto \emptyset\}$$

The proof of the weakest preconditions are relatively straightforward, and mostly follows symbolic execution. The primary proof effort is to consolidate the fractional views to unify and update the model state at the appropriate times, to stay consistent with the invariant.

With the weakest preconditions in hand, we can finally apply the adequacy theorem. As for the retransmit example, we can allocate all the necessary ghost state up front, and use it when allocating the invariant. From this, it is trivial to observe that the assumptions of the weakest preconditions follow directly from the initial resources of the adequacy theorem.

With the refinement and the *AlwaysHolds* property in hand, along with the proof of the liveness property at the model level, we can derive the liveness property for the implementation of the Stenning protocol, completing the proof.

5 Inner Workings of Fairneris

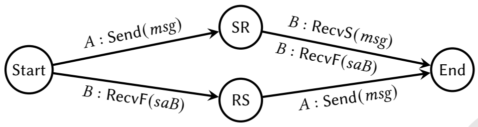
This section explains *how* the Fairneris logic is constructed, and how its adequacy theorem is established. As such, it is the most technical section of the paper, and is not necessary to read to understand how Fairneris is *used*.

The adequacy theorem of Fairneris, Theorem 3.1 establishes that given any trace *ex* of the program, we obtain a trace *utr* in the user model $\mathcal{U}$ such that $ex \preccurlyeq_{\xi_U} utr$. As we briefly explained in §1 the trace *utr* is not constructed directly. Rather, the program logic, behind the scene, constructs a trace *ftr* in the model $\mathsf{Fuel}(\mathcal{U} \ltimes \mathcal{N}et)$, and then extracts *utr* out of *ftr*. Below, we will first explain the details of the construct $\mathsf{Fuel}(\mathcal{U} \ltimes \mathcal{N}et)$ and how the trace *utr* is obtained from *ftr*. Afterwards, we explain how the program logic of Fairneris constructs the trace *ftr*. While we keep these explanation self-contained, we will mainly focus on the innovations that were necessary for this process compared to Fairis and the Trillium framework.

5.1 Obtaining User Model Trace

Recall the user model of the retransmit example in Figure 1 from §2. The transitions in user models are annotated with two pieces of information a role and a an action. (In general, user model transition could also only have a role and no action corresponding to state changes that do not correspond to any network action.) These two pieces of information, roles and actions, are respectively there to facilitate hiding details of the scheduler and those of the network, respectively. To better understand, and appreciate the subtlety of obtaining a trace in $\mathcal{U}$ from a trace of $\mathsf{Fuel}(\mathcal{U} \ltimes \mathcal{N}et)$, let us first for now ignore the Fuel construction and focus on the difference between traces of $\mathcal{U} \ltimes \mathcal{N}et$ and $\mathcal{U}$.

The Subtle Relation Between Traces of $\mathcal{U} \ltimes \mathcal{N}et$ and $\mathcal{U}$. Let use consider a very simple program consisting of two nodes each with a single thread where one performs a single **send** operation and the other performs a single **recv** operation. This distributed program obviously terminates after performing exactly one send and one receive operation (which may fail or succeed). Let us consider the following simple user model for this distributed program:



$msg \triangleq (saA, \texttt{"x"}, saB)$

As expected, the use model above has finitely many traces with the maximum length of two. However, the model $\mathcal{N}et$, and therefore also $\mathcal{U} \ltimes \mathcal{N}et$, has infinitely many states and infinitely many traces of arbitrary length including infinite traces. This is because $\mathcal{N}et$ must reflect all the non-deterministic behavior of the network. Below, are a few of these traces (we omit states):

$$\cdot \xrightarrow{\mathsf{Send}(msg)} \cdot \xrightarrow{\mathsf{Deliver}(msg)} \cdot \xrightarrow{\mathsf{RecvS}(msg)} \cdot \tag{1}$$

$$\cdot \xrightarrow{\mathsf{Send}(msg)} \cdot \xrightarrow{\mathsf{Dup}(msg)} \cdot \xrightarrow{\mathsf{Deliver}(msg)} \cdot \xrightarrow{\mathsf{Deliver}(msg)} \cdot \xrightarrow{\mathsf{RecvS}(msg)} \cdot \tag{2}$$

$$\cdot \xrightarrow{\mathsf{Send}(msg)} \cdot \xrightarrow{\mathsf{Dup}(msg)} \cdot \xrightarrow{\mathsf{Deliver}(msg)} \cdot \xrightarrow{\mathsf{Drop}(msg)} \cdot \xrightarrow{\mathsf{RecvS}(msg)} \cdot \tag{3}$$

$$\cdot \xrightarrow{\mathsf{Send}(msg)} \cdot \xrightarrow{\mathsf{Dup}(msg)} \cdot \xrightarrow{\mathsf{Deliver}(msg)} \cdot \xrightarrow{\mathsf{Dup}(msg)} \cdot \xrightarrow{\mathsf{RecvS}(msg)} \cdot \xrightarrow{\mathsf{Drop}(msg)} \cdot \tag{4}$$

$$\cdot \xrightarrow{\mathsf{Send}(msg)} \cdot \xrightarrow{\mathsf{Dup}(msg)} \cdot \xrightarrow{\mathsf{Deliver}(msg)} \cdot \xrightarrow{\mathsf{RecvS}(msg)} \cdot \xrightarrow{\mathsf{Dup}(msg)} \cdot \xrightarrow{\mathsf{Deliver}(msg)} \cdot \xrightarrow{\mathsf{Dup}(msg)} \cdot \xrightarrow{\mathsf{Drop}(msg)} \cdots \tag{5}$$

In fact all of these traces correspond to the following $\mathcal{U}$ trace:

$$\mathsf{Start} \xrightarrow{\mathsf{Send}(msg)} \mathsf{SR} \xrightarrow{\mathsf{RecvS}(msg)} \mathsf{End} \tag{6}$$

Observe the difference between the $\mathcal{U}$ trace (6) and traces (1–5). The difference is in actions performed by the network (no program involvement): dropping, delivering, and duplicating msg. We call these system actions. There are two kinds of system actions: relevant ones, those after which there is a program step, and irrelevant ones, those that appear at the end of the trace. In (1), (2), and (3) all system actions are relevant while in (4) and (5) the system actions that happen after $\mathsf{RecvS}(msg)$ are irrelevant; infinitely many in case of (5).

Refinement Tower. The refinement between a program execution ex and a trace utr of the user model $\mathcal{U}$ is defined in several steps, which are depicted in Figure 4. Going bottom up, the Trillium adequacy theorem ensures there is a lockstep forward simulation between the program execution ex and a trace ftr of $\mathsf{Fuel}(\mathcal{U} \ltimes \mathcal{N}et)$, where the Fuel-construction, described in Fairis [34], provides finite stuttering and scheduler fairness preservation, in the sense that if ex is fair (w.r.t. thread ids), then ftr is fair w.r.t. *roles*. This step will be discussed in §5.4.

Removing finite stutter from ftr yields a trace jtr of the "joint model" $\mathcal{U} \ltimes \mathcal{N}et$. Extracting a trace utr of the user model $\mathcal{U}$ from jtr is done in two steps: first the trailing network step (irrelevant system actions) are removed in a process we call *trimming*, resulting in a trace ttr of $\mathcal{U} \ltimes \mathcal{N}et$.

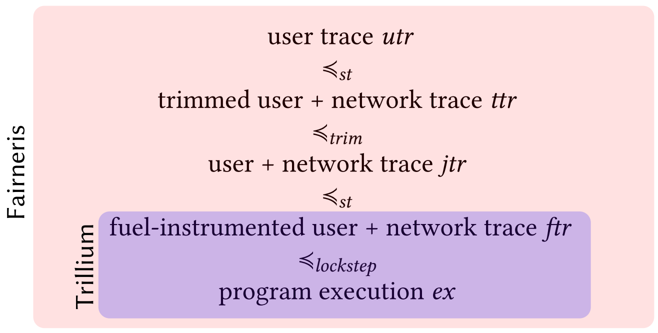

user trace utr
$\preccurlyeq_{st}$
trimmed user + network trace ttr
$\preccurlyeq_{trim}$
user + network trace jtr
$\preccurlyeq_{st}$
fuel-instrumented user + network trace ftr
$\preccurlyeq_{lockstep}$
program execution ex
Fairneris
Trillium

Fig. 4. The trace refinements involved in Fairneris

Finally, the resulting trimmed trace ttr is projected onto $\mathcal{U}$, yielding the user model trace utr, that the user can use to reason about the distributed system — during this step we also remove all the relevant system actions.

The important properties of all these refinements is that:

(i) They preserve (scheduler and network) fairness in the upward direction (according to Figure 4) so that if the execution ex is fair, then so is utr, and
(ii) They all downwards-preserve liveness properties of interest, such as LTL formulas of the form $\mathbb{F}P$, where P only depends on the current network action.

For example, this allows us to conclude that the program of Figure 1 eventually sends the "done!" message from the fact that the model does.

Here, we describe the two types of refinement that are involved in the light pink region of Figure 4 (the outer square), and how they are obtained in a generic way without any additional proof burden on the user.

Trimming Refinement. The traces we consider contain both transitions coming from the user model $\mathcal{U}$ and transitions coming from the network model $\mathcal{N}et$. The first step to extract the underling user trace is to remove all trailing network steps, a process we call *trimming*. The reason is that, in a trimmed trace, there are necessarily finitely many network transitions before the next user transition. This allows us to define a productive function that removes network transitions, as described below.

Among the three kinds of refinement we use, trimming is the worst-behaved with respect to preservation of properties, as it is very asymmetric. For example, if $tr_1 \preccurlyeq_{trim} tr_2$, and if $tr_2 \Vdash \mathbb{F}P$, then $tr_1 \Vdash \mathbb{F}P$, but the converse is false. However, this property is sufficient for property (ii) above. Fairness preservation is proved with an ad-hoc reasoning which depends on the nature of the atomic predicates involved.

Stutter Refinement. According to the standard definition, the trace t_1 *stutter-refines* t_2 if t_2 can be obtained from t_1 by removing finite stutter, where a finite stutter is a finite sequence of repeated states. For instance, the sequence $122333\cdots$ stutter-refines $123\cdots$. In our setting however, where the two traces are equipped with states and labels of different nature, the definition sketched above is not applicable. Our solution is to consider a *heterogeneous* notion of stutter-equivalence, between traces with different states S_1, S_2 and labels L_1, L_2. This notion of refinement is parameterized by a map $\pi_S : S_1 \to S_2$ between states and a partial map $\pi_L : L_1 \rightharpoonup L_2$ between labels. The idea is that a step $s_1 \xrightarrow{\ell} s_1'$ in a trace t_1 is a stutter step iff $\pi_S(s_1) = \pi_S(s_2)$ and if $\pi_L(\ell)$ is not defined. Formally, it is defined as follows, where only the rightmost rule can be used infinitely many times in a row.[3]

$$\frac{\pi_S(s_1) = s_2}{\langle s_1 \rangle \preccurlyeq_{st} \langle s_2 \rangle} \qquad \frac{\pi_S(s_1) = \pi_S(\mathsf{hd}(t_1)) = \mathsf{hd}(t_2) \quad \pi_L(\ell_1) = \bot \quad t_1 \preccurlyeq_{st} t_2}{s_1 \xrightarrow{\ell_1} t_1 \preccurlyeq_{st} t_2} \qquad \frac{\pi_S(s_1) = s_2 \quad \pi_L(\ell_1) = \ell_2 \quad t_1 \preccurlyeq_{st} t_2}{s_1 \xrightarrow{\ell_1} t_1 \preccurlyeq_{st} s_2 \xrightarrow{\ell_2} t_2}$$

In practice, we only have the trace t_1 and we wish to obtain a trace t_2 such that $t_1 \preccurlyeq_{st} t_2$. This is possible when the set of traces with states in S_1 and labels in L_1 is equipped with a "potential" function $\Psi : \mathbf{Trace}(S_1, L_1) \to \mathbb{N}$ that decreases along stutter transitions.

We use this technique twice: once for removing the fuel instrumentation, as in Fairis, and once for obtaining the user trace from the trimmed trace ttr. In the latter case, network transitions are the stutter transitions, as they cannot modify the user state — see the construction of $\mathcal{U} \ltimes \mathcal{N}et$ explained below. To define the potential Ψ, we rely on the fact that the traces we consider are trimmed and satisfy scheduling fairness, which means that there can only be finitely many network transitions in a row. Thus, we define $\Psi(ttr)$ to be the number of network transitions before the first program step.

To reason about property preservation along stutter refinement, we defined the predicate

$$P \sim_{st} Q \iff \forall tr_1, tr_2.\ \ tr_1 \preccurlyeq_{st} tr_2 \Rightarrow (tr_1 \Vdash P \Leftrightarrow tr_2 \Vdash Q)$$

which, intuitively, states that P corresponds to Q when going through a stutter refinement in either direction. It is a congruence with respect to LTL modalities we use for example, $P \sim_{st} Q$ implies $\mathbb{F}P \sim_{st} \mathbb{F}Q$, and $\mathbb{G}P \sim_{st} \mathbb{G}Q$. However, for point predicates P and Q, for example those which only depend on the current label of the trace, it is not obvious how to relate them under $\sim_{st}$. Nevertheless, the $\mathbb{F}$ modality alleviates this issue because we can show the following:

$$(\forall \ell,\ P(\ell) \Leftrightarrow (\pi_L(\ell) \neq \bot \wedge Q(\pi_L(\ell)))) \implies \mathbb{F}P \ \sim_{st}\ \mathbb{F}Q$$

As it turns out, this simpler result is sufficient to prove the properties (i) and (ii) for $\preccurlyeq_{st}$.

5.2 Concrete Operational Semantics of Distributed Systems

The Trillium framework [34], just like the Iris framework it is based on does not fix the programming language — hence, both Fairis and Fairneris can be constructed on top of it. In practice, this means that the Trillium framework introduces its own notion what a programming language is. This allows instantiations to instantiate Trillium by constructing an instance of this notion. In this work, we have made a simple adjustment to this notion of a programming language: we have introduced *actions*. This simple addition allows us to appropriately abstract network's behavior as we will discuss below.

A programming language for Trillium is defined fixing a set of terms, *Term*, a set of states, *State*, a set of locales, *Locale*, a set of user actions, **Usract**, and a set of system actions, **Sysact**, together with the following two relations for program steps and system steps:

$$\begin{aligned} \to_{\mathsf{prg}} &\subseteq (\mathit{Term} \times \mathit{State}) \times \mathbf{Usract}^? \times (\mathit{Term} \times \mathit{State} \times \mathit{Term}^\star) \\ \to_{\mathsf{sys}} &\subseteq \mathit{State} \times \mathbf{Sysact} \times \mathit{State} \end{aligned}$$

Terms are program expressions (AnerisLang expressions in case of AnerisLang). States describe the state of the entire system (for AnerisLang: a memory heap and a map of socket handles for each node, together with a so-called message soup of messages in transit). Locales are conceptual locations where programs run (a pair of an IP address together with a thread id for AnerisLang). A tuple $(e, \sigma, \alpha^?, e', \sigma', \vec{e}_f) \in \to_{\mathsf{prg}}$ describes a single step of the program reducing the term e to the term e' and updating the state from σ to σ' while forking threads in the sequence $\vec{e}_f$ of expressions — in AnerisLang $\vec{e}_f$ is only non-empty for the fork operation. The α label is optional (denoted by

[3]This definition should be understood as an inductive relation nested in a co-inductive relation.

?), to allow for internal transitions of the program, which do not involve the network. A tuple $(\sigma, \beta, \sigma') \in \rightarrow_{\text{sys}}$ describes systems' autonomous updates; dropping, delivering, or duplicating in-transit messages in AnerisLang.

Based on these, Trillium defines so-called configurations as a pair of a thread pool (a mapping from locales to terms) and a state, $\mathit{Cfg} \triangleq \mathit{ThPool} \times \mathit{State}$, and defines a global step relation

$$\rightarrow_{glob} \quad \subseteq \quad \mathit{Cfg} \times ((\mathit{Locale} \times \mathbf{Usract}^?) \cup \mathbf{Sysact}) \times \mathit{Cfg}$$

The relation $\rightarrow_{glob}$ is defined by non-deterministically picking a locale and running it for one step, or by non-deterministically picking a system step. This describes all behaviors of the system including fair and unfair behaviors of the scheduler and the network.

For AnerisLang, the program steps and system steps are defined in the standard way for an OCaml-like programming language with UDP-like networking primitive that it is [20]. Here, we only describe the state in AnerisLang as it is relevant to the discussion in this paper. See Krogh-Jespersen et al. [20] for more details. In AnerisLang, the global state of a distributed system is of the form $((\mathcal{S}, \mathcal{M}), \mathcal{H})$, where $\mathcal{H}$ is a map associating each node $ip \in Ip$ with a program heap $h = \mathcal{H}(ip)$. The message soup $\mathcal{M}$ is a multiset of messages in transit. The socket state map $\mathcal{S}$ is a mapping from IP addresses to *local socket state* S, which itself is a map from *socket handlers* to pairs (skt, buf) of the receive buffer buf, and the *socket* itself skt, which contains the socket address $skt.sa$. Additionally, there are some well-formedness conditions, for example: $\mathcal{S}[ip] = S$ and $S[sh] = (skt, buf)$ imply that the IP address of the socket address of skt is equal to ip.

5.3 Model Abstraction for Network Traces

Fairneris adequacy, Theorem 3.1, states the existence of a refinement between fair executions of the distributed system and traces of the user provided model $\mathcal{U}$. As we will see in §5.4, the adequacy theorem of the underlying Trillium logic constructs a refinement between the distributed system and a more complicated model, that contains both the user model $\mathcal{U}$ and an abstract description of the network. This section explains how this abstract network is combined with the user model $\mathcal{U}$, and how the refinement $\preccurlyeq_{\xi_U}$ is defined.

Synchronized Product. The global model we use is the combination (obtained using the operation $\ltimes$) of the user model, chosen by the user, and the network model, which is fixed. For the sake clarity, we first explain how operation $\ltimes$ is defined in general for an arbitrary instantiation of Trillium. That is, we define the composition of a user model with a *system model*, which abstracts the *system* from the previous section. The two notions of model differ slightly, in that only user models keep track of *roles*, as the fairness of the network and of the scheduler are of different nature.

Definition 5.1 (Enviroment model). Consider a Trillium language instantiation with set of states *State* and set of system actions **Sysact**. A *system model* is a LTS $\mathcal{N}$ labeled **Sysact**, together with a relation

$$\text{env\_match} \quad \subseteq \quad \mathit{State} \times \mathcal{N}$$

and a map

$$\text{appSys} : \mathcal{N} \rightarrow \mathbf{Sysact} \rightarrow \mathcal{N}$$

The relation env_match expresses that the concrete state of the system $\sigma \in \mathit{State}$, refines an abstract state $n \in \mathcal{N}$. The appSys specifies how the system state evolves when the system makes an autonomous action, *i.e.*, it reflects the relation $\rightarrow_{\text{sys}}$. The above data also needs to satisfy some properties which we elude here.

The purpose of the appSys function is to preserve the refinement env_match between the distributed system and the abstract model when the network takes an autonomous step. This function is fixed because system steps are not under the control of the program, and so should be invisible to the user of the logic. A *user model* is essentially an LTS with specific labels:

Definition 5.2 (User model). Given a Trillium language instantiation with set of user actions **Usract**, a *user model* $\mathcal{U}$ is the data of a set **role** of *roles*, together with an LTS $\mathcal{N}$ labeled with pairs $(\rho, \alpha^?) \in \textbf{role} \times \textbf{Usract}^?$.

The motivation for these two definitions is that we can combine them together to yield a model that can be "tightly" refined by a distributed system, in the sense that their states are related by a concrete relation that is amenable to be prove refinements with respect to using a program logic such as Trillium.

Definition 5.3. Given a user model $\mathcal{U}$ and a system model $\mathcal{N}$ over the same Trillium language instantiation, their *synchronized product* $\mathcal{U} \ltimes \mathcal{N}$ is an LTS labeled with $(\textbf{role} \times (\textbf{Usract}^?)) \cup \textbf{Sysact}$, whose states are pairs (m, n) of states of $\mathcal{U}$ and of $\mathcal{N}$, respectively, together with the following transitions:

$$\frac{m_1 \xrightarrow{(\rho,\perp)} m_2}{(m_1, n) \xrightarrow{(\rho,\perp)} (m_2, n)} \qquad \frac{m_1 \xrightarrow{(\rho,\alpha)} m_2 \quad n_1 \xrightarrow{\alpha} n_2}{(m_1, n_1) \xrightarrow{(\rho,\alpha)} (m_2, n_2)} \qquad \frac{n_1 \xrightarrow{\beta} n_2}{(m, n_1) \xrightarrow{\beta} (m, n_2)}$$

They correspond, respectively, to a step of the user model without any actions, a step of the user model that has an action which triggers a step in the system model, and an autonomous step of the system.

In summary, internal program steps, *i.e.* when the action is not present, $\alpha^? = \perp$, do not synchronize with the system, and neither do the autonomous system steps with the program. The two sides synchronize through the program actions, which correspond to both a program step and a system step.

5.3.1 The Abstract Network Model.

In Fairneris, we instantiate the system model with the *network model* $\mathcal{N}et$. This network model is more abstract than the one used in the operational semantics of AnerisLang, as it eschews socket handlers. The states $(\mathsf{sm}, \mathcal{M})$ of $\mathcal{N}et$ are simply maps sm from socket addresses to lists of messages, representing the receive buffers, as well as the global message soup $\mathcal{M}$. The labels are the system actions of AnerisLang, described in §5.2. The transitions of this LTS are defined in a very similar way as the low-level network semantics. Thus, we omit them in the paper. The appSys function is defined in the obvious way, for example the case of message deliveries is the following:

$$\mathsf{appSys}((\mathsf{sm}, \mathcal{M}), \mathsf{Deliver}(msg)) \;=\; (\mathsf{sm}[sa := msg{::}\mathsf{sm}[sa]], \mathcal{M} \setminus \{msg\})$$

where *sa* is the destination socket address of *msg*, the message being delivered, and :: is the *cons* operation on lists.

To build the refinement between the program and the model, we maintain a relationship env_match between the two representations of the network: an AnerisLang network $(\mathcal{S}, \mathcal{M}')$ refines a model network $(\mathsf{sm}, \mathcal{M}')$ when

$$\mathcal{M} = \mathcal{M}' \tag{7}$$

$$\mathcal{S}[ip] = S \;\wedge\; S[sh] = (skt, buf) \implies \mathsf{sm}[skt.sa] = buf \tag{8}$$

$$\mathsf{sm}[sa] = buf \implies \exists sh, skt.\; \mathcal{S}[ip][sh] = (skt, buf) \;\wedge\; skt.sa = sa \tag{9}$$

Intuitively, the message soups have to match exactly, and the receive buffers have to contain the same messages. Note that there could be an empty receive buffer in the AnerisLang network state that is not allocated at all in the model network state, this allows $\rightarrow_{\text{prg}}$ transitions such as creating a new socket to *not* be reflected in the abstract model.

5.4 Model of the Logic

We conclude this section with a discussion of how the Fairneris logic is defined. First we explain the concept of *trace interpretation* [34] and define that of Fairneris. This allows us to explicate the definitions of the predicates $\mathsf{wp}_{\mathcal{E}}\, e\, \langle\Phi\rangle$ and $\mathsf{mu}_{\mathcal{E}}^{\zeta;\alpha^?}$, that were explained in §3, and which manipulate this trace interpretation. Finally, we sketch how the first refinement in Figure 4 is obtained using the Trillium logic.

Trace Interpretation. The general weakest precondition of Trillium is parameterized by a so-called *trace interpretation* predicate, an Iris predicate relating traces of the program to the traces of the model. (This is a generalization of what Iris calls a state interpretation.) We write $S(\tau, \kappa)$. In our instantiation of Trillium with AnerisLang, the arguments τ and κ are respectively, a finite execution of the program (a sequence of *Cfg*'s to be more precise), and a finite trace of the global model $\mathsf{Fuel}(\mathcal{U} \ltimes \mathcal{N}et)$. These are the histories of the program and of the model, and are intuitively prefixes of the full executions *ex* and user model traces *utr* we considered above. Intuitively, $S(\tau, \kappa)$ states that the current state of the program is $last(\tau)$, the current state of the model is $last(\kappa)$, and that they match, in the sense determined by the person instantiating Trillium who picks the trace interpretation relation. Note that, since Trillium is designed to build lockstep simulations, the two histories τ and κ must always have the same length. For the Fairneris instantiation of Trillium, the trace interpretation is defined as follows:

$$S(\tau, \kappa) \;\; := \;\; \mathsf{traces\_match}(\tau, \kappa) \, * \, \mathsf{aneris\_interp}(\tau) \, * \, \mathsf{model\_interp}(last(\tau), last(\kappa))$$

where aneris_interp is the existing Aneris state/trace interpretation, reflecting the state of node heaps, sockets, and the message soup in Iris resources. The traces_match predicate is a straightforward modification of that same predicate from Fairis, adding support for actions in the operational semantics of the programming language and in the model. Here, traces_match also adds the requirement that actions must match exactly in the two traces, allowing us to transport knowledge about the network actions back-and-forth between the distributed system and the abstract model ($\mathcal{U} \ltimes \mathcal{N}et$). Finally, the relation $\mathsf{model\_interp}((tp, \sigma), \tilde{m})$ asserts that the current state of the model is $\tilde{m}$, and that it is well-formed. The state $\tilde{m} \in \mathsf{Fuel}(\mathcal{U} \ltimes \mathcal{N}et)$ can be decomposed into three parts, that are each treated differently:

- The fuel instrumentation is handled as in Fairis [34]
- The state m of the user model $\mathcal{U}$ is owned by the predicate S[4]
- The network part $\mathsf{n} \in \mathcal{N}et$ is novel: the trace interpretation S asserts $\mathsf{env\_match}(\sigma, \mathsf{n})$ holds.

The Single Step Weakest Precondition. One of the main innovation of Fairneris over Fairis in terms of their program logics is the logical rules governing the *single step weakest precondition* $\mathsf{wp}_{\mathcal{E}}\, e\, \langle\Phi\rangle$ (abbreviated SSWP), which was introduced by Fairis, and its companion, the *model update* $\mathsf{mu}_{\mathcal{E}}^{\zeta;\alpha^?}$, which we have introduced in this paper. In Fairis[34], SSWP was intuitively defined as follows: during a single step of the program, given the current trace interpretation $S(\tau, \kappa)$, we can update the ghost state in such a way that the trace interpretation for the extended histories $S(\tau \xrightarrow{\zeta} (tp', \sigma'), \kappa \xrightarrow{\ell} m')$holds. As such, logically, and in terms of user experience, updating the ghost state for the program and for the model $\mathcal{M}$ had to be done at the same time.

[4]For readers familiar with Iris, S asserts ownership of the authoritative part of the resource algebra $\textsc{Auth}(\mathrm{Ex}(\mathcal{U}))$

Trillium SSWP

$$\begin{aligned}\mathsf{wp}_{\mathcal{E}}\; e\;\langle\Phi\rangle \triangleq \big(e \notin \mathit{Val} * \forall \tau,\kappa,\sigma,K,\mathsf{tp}.\; \mathit{last}(\tau) = (\mathsf{tp}[\zeta \mapsto K[e]],\sigma) * S(\tau,\kappa) \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \\ \mathit{reducible}(e,\sigma) * \triangleright \forall \alpha^?, e', \sigma', \vec{e}.\; (e,\sigma) \xrightarrow{\alpha^?}_{\mathsf{prg}} (e',\sigma';\vec{e}) \mathrel{-\!\!*} {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \\ S'(\tau \xrightarrow{(\zeta,\alpha^?)} (\mathsf{tp}[\zeta \mapsto K[e']],\sigma'),\kappa,\zeta,\alpha^?) * \Phi\;(e',\alpha^?) * \vec{e} = \epsilon\big)\end{aligned}$$

Trillium Model Update

$$\mathsf{mu}_{\mathcal{E}}^{\zeta;\alpha^?}\; P \triangleq \big(\forall \tau,\kappa.\, S'(\tau,\kappa,\zeta,\alpha^?) \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} \exists \mathsf{m} \in \mathcal{M},\rho.\;\; \mathit{last}(\kappa) \xrightarrow{\rho} \mathsf{m} * S(\tau,\kappa \xrightarrow{\rho} \mathsf{m}) * P\big)$$

System weakest precondition

$$\mathsf{system\_wp} \;\triangleq\; \forall \tau,\kappa,\beta,\mathit{tp},\sigma.\;\; \mathit{last}(\tau) \xrightarrow{\beta}_{\mathit{sys}} (\mathit{tp},\sigma) \;*\; S(\tau,\kappa) \mathrel{-\!\!*} {}^{\top}\!\Rrightarrow^{\top} \exists \mathsf{m},\ell.\; S(\tau \xrightarrow{\beta} (\mathit{tp},\sigma),\kappa \xrightarrow{\ell} \mathsf{m})$$

Fig. 5. Definitions of basic constructs

The new definition of the SSWP, and that of model update, given in Figure 5, decouple these two concerns. Here, make use of a so-called *torn trace interpretation* $S'(\tau,\kappa,\zeta,\alpha^?)$, which expresses that the program history τ is one step ahead of the model history κ, for a step that is labeled by $(\zeta,\alpha^?)$. Formally, it is defined as:

$$\begin{aligned}S'(\tau,\kappa,\zeta,\alpha^?) \;\;\triangleq\;\; \exists \tau',\mathit{tp},\;\sigma.\;\;\; \tau \;=\; \tau' \xrightarrow{(\zeta,\alpha^?)} (\mathit{tp},\sigma) \;* \\ \mathsf{traces\_match}(\tau',\kappa) \;*\; \mathsf{aneris\_interp}(\tau') \;*\; \mathsf{model\_interp}(\mathit{last}(\tau'),\mathit{last}(\kappa))\end{aligned}$$

The Aneris part of the ghost state corresponds to extended trace τ of the program, whereas the current model history corresponds to the immediate prefix τ' of the program history. Thus, the SSWP moves the program execution by one step, tearing the relationship between the traces of the program and the the model. The model update then moves the model trace one step so that they match again. This explains the inference rule WP-STEP, which decomposes a weakest precondition into (1) an SSWP handling the next step of the program, (2) a model update handling the next step of the model, and (3) a weakest precondition handling the remainder of the steps of both the program and the model.

The System Weakest Precondition. The intuitive reason wp $e\;\{\Phi\}$ guarantees safety of the program e is that it states that every step of the *program* is safe, in the sense that the program should not crash (should not get stuck), and that it preserves the trace interpretation. In the case of distributed systems that is not enough, as the *network* also takes *autonomous* steps. These steps need to not only should not cause a crash, but they must also preserve the trace interpretation.

The solution in Trillium [34] was to introduce the *system weakest precondition* (system WP), which simply states that all steps $\sigma \rightarrow_{\mathit{sys}} \sigma'$ of the network preserve the trace interpretation, as shown in Figure 5. Unlike weakest preconditions that are proved by the users of a program logic like Fairneris based on the Trillium framework, the system WP is proved once and for all by the person instantiating Trillium, and does not appear at all to the user.

In the case of Fairneris this step is more difficult than for the previous Aneris instantiation of Trillium by Timany et al. [34], as we additionally need to prove that each network step refines the corresponding step in the abstract network model $\mathcal{N}et$. This is one of the places the gap between the concrete network and abstract model of the network needs to be bridged, by proving that properties (7–9) are preserved.

	Fairneris (this work)	Aneris [20]	Disel [30]	Grove [31]	Igloo [32]	IronFleet [16]	Verdi [35]
Higher-Order	✓	✓	✓	✓	✓	✓	✗
Concurrency	✓	✓	✗	✓	✓	✗	✗
Safety properties	✓	✓	✓	✓	✓	✓	✓
Liveness properties	✓	✗	✗	✗	✗	✓*	✗
Refinement-based	✓	✓†	✗	✗	✓	✓	✗‡
Foundational	✓	✓	✓	✓	✗§	✗	✓¶
Crash Recovery & Reconfiguration	✗	✗	✗	✓	✗	✗	✗

*Requires a special program shape: a main event loop.

†The version by Timany et al. [34] is refinement-based; albeit only for safety properties.

‡Refinement between programs; initially programs are written against an idealized network model and then automatically refined to programs that assume a less reliable network model.

§Meta theory proven correct in the Isabelle proof assistant [1]; programs proven correct in Verifast [18] and Negini [12].

¶Relies on a not-so-small shim to run programs written in Verdi's DSL.

Table 1. Comparison of different approaches to reasoning about distributed systems' implementations.

The Lockstep Refinement Using Trillium. Fairneris is based on Trillium[34], and thus inherits its weakest precondition $\mathrm{wp}^{\zeta}_{\mathcal{E}}\; e \;\{\Phi\}$. The first step of the proof of Fairneris adequacy (Theorem 3.1) is to apply Trillium's adequacy, Theorem 5.4. Below, we will present and explain this theorem and sketch how it is used in proving the adequacy theorem of Fairneris.

Theorem 5.4 (Trillium adequacy). *Let e be a program, σ a program state, ζ the locale of e, and Φ an Iris predicate on values. Let $\mathsf{m} \in \mathcal{M}$ be a model state and ξ a relation on finite traces of the program and the model. Let $(\{\zeta \mapsto e\}, \sigma)$ be the initial configuration of the program. If $\xi((\{\zeta \mapsto e\}, \sigma), \mathsf{m})$ holds for the initial singleton traces, and furthermore we have*

$$\Rrightarrow_{\top} S(c, \mathsf{m}) * wp^{\zeta}_{\top}\; e \;\{\Phi\} * \mathit{AlwaysHolds}_{\top}(\xi, c, \mathsf{m})$$

then $(\{\zeta \mapsto e\}, \sigma) \precsim_{\xi} \mathsf{m}$ holds in the meta-logic.

The predicate $(\{\zeta \mapsto e\}, \sigma) \precsim_{\xi} \mathsf{m}$ states that there exists a lockstep forward simulation that preserves the relation ξ, and which relates $(\{\zeta \mapsto e\}, \sigma)$ and m. The $\mathit{AlwaysHolds}_{\top}$ predicate is analogous to the corresponding predicate in Theorem 3.1. The refinement $\preccurlyeq_{lockstep}$ at the bottom of Figure 4 is obtained from $\precsim_{\xi}$ using a general result stating that such simulation relations induce a refinement relation between possibly-infinite traces [34]. All that remains is to apply Theorem 5.4 with of $\mathcal{M} := \mathsf{Fuel}(\mathcal{U} \ltimes \mathit{Net})$, where $\mathcal{U}$ is the model chosen by the user of Fairneris and with a well-chosen relation ξ, which incorporates the user chosen relation ξ_U. The Fuel construction, which is taken essentially verbatim from Fairis [34], together with ξ ensure that $\precsim_{\xi}$ preserves fairness. We remark that this is more difficult than for the other refinement relations, because fairness of the program execution is defined in terms of *locales*, whereas the fairness for traces of $\mathsf{Fuel}(\mathcal{U} \ltimes \mathit{Net})$ is defined in terms of the *roles* of $\mathcal{U}$.

6 Related and Future Work

Verification of Distributed Systems' Implementations. We have already discussed the works most closely related to ours, *i.e.*, Aneris [20], Disel [30], Grove [31], Igloo [32], IronFleet [16], and Verdi [35], in §1. Table 1 gives a summary of comparison of these systems.

Regarding Igloo, we remark that it proves correctness of distributed systems by establishing a rather weak refinement relation between the program and a transition system which only establishes that the program and the transition system perform the same I/O operations. The weakness of Igloo's refinement relation enables great flexibility with respect program structure thus enabling it to reason about concurrent programs that use higher-order programming features which are not written as simple event loops. However, it is not clear how the methodology of Igloo [32] can be extended to support reasoning about liveness properties.

The approach of Grove [31] to verifying distributed systems is very close to that of Aneris [20] upon which we have based our work — Grove is also based on the Iris framework [19]. It pushes the limits of this methodology by verifying (safety properties of) crash recovery (some nodes may spontaneously fail and restart) and reconfiguration (nodes can dynamically join or leave the distributed system). An interesting and important future work is to prove liveness properties of such distributed systems. Indeed, one of the main reasons for implementing crash recovery and dynamic reconfiguration in distributed systems is to improve availability. We believe that the Fairneris methodology developed here should also scale to supporting reasoning about these more advanced features of distributed systems.

Verification of Liveness Proeprties of Distributed Protocols. Several works [4, 5, 11, 24, 25, 27, 28, 36] have considered proving liveness properties of distributed systems at the protocol level by studying these protocols as state transition systems at various levels of detail. Most of these approaches focus on specifically proving correctness of consensus protocols. They often [11] even require the program to be constructed in terms of rounds of reaching consensus. Even though these approaches would be able to verify the stenning protocol as we have, they do net seem able to handle our simpler retransmit example in §2. In many cases [4, 5, 24, 25, 27] these works verify liveness properties by reducing liveness properties to safety properties which they then automate. Compared to our approach of verifying liveness properties of concrete implementations, these approaches scale better, *e.g.*, to proving liveness in the presence of byzantine faults [4, 5, 24, 28]. Proving liveness of distributed systems in the presence of byzantine faults is another important and interesting future work. We believe that the state-transition-systems style protocols studied by these works are interesting candidates to consider as high-level models in Fairneris. This would allow us to scale Fairneris's approach to reasoning about liveness in the presence of byzantine faults.

Verification of Stenning Protocol. Both safety and liveness properties of the Stenning protocol have been verified at different levels of details [8, 9, 15, 26, 29]. Among these, only Compton [8] considers a concrete implementation, in OCaml, on top of UDP sockets. However, Compton [8] only proves safety properties, whereas we prove both safety and liveness of the protocol.

7 Conclusion

We have presented Fairneris, the first higher order concurrent separation logic that is capable of reasoning about both the safety and liveness of concrete implementations of distributed systems, by following the methodology of Fairis and Trillium, *i.e.*, via constructing and exploiting a refinement relation between the distributed program and an abstract model of the distributed algorithm it implements. Using this method, the program logic handles what it does best — the complex semantics of the language and the safety properties — while the liveness reasoning happens on a hight level representation of the system.

References

[1] 2002. *Isabelle/HOL*. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-45949-9

[2] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. 1975. Some constraints and tradeoffs in the design of network communications. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles* (Austin, Texas, USA) *(SOSP '75)*. Association for Computing Machinery, New York, NY, USA, 67–74. https://doi.org/10.1145/800213.806523

[3] Bowen Alpern and Fred B. Schneider. 1985. Defining liveness. *Inform. Process. Lett.* 21, 4 (1985), 181–185. https://doi.org/10.1016/0020-0190(85)90056-0

[4] Idan Berkovits, Marijana Lazić, Giuliano Losa, Oded Padon, and Sharon Shoham. 2019. Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 245–266.

[5] Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazić, Pierre Tholoniat, and Josef Widder. 2022. Holistic Verification of Blockchain Consensus. In *36th International Symposium on Distributed Computing (DISC 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 246)*, Christian Scheideler (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:24. https://doi.org/10.4230/LIPIcs.DISC.2022.10

[6] James Adam Cataldo. 2006. *The Power of Higher-Order Composition Languages in System Design.* Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-189.html

[7] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '07)*. Association for Computing Machinery, New York, NY, USA, 398–407. https://doi.org/10.1145/1281100.1281103

[8] Michael Compton. 2005. Stenning's Protocol Implemented in UDP and Verified in Isabelle. In *Theory of Computing 2005, Eleventh CATS 2005, Computing: The Australasian Theory Symposium, Newcastle, NSW, Australia, January/February 2005 (CRPIT, Vol. 41)*, Mike D. Atkinson and Frank K. H. A. Dehne (Eds.). Australian Computer Society, 21–30. http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV41Compton.html

[9] Benedetto Lorenzo Di Vito. 1982. *Verification of communications protocols and abstract process models.* Ph. D. Dissertation. AAI8227642.

[10] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *ACM Trans. Program. Lang. Syst.* (2021). https://doi.org/10.1145/3477082

[11] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 400–415. https://doi.org/10.1145/2837614.2837650

[12] Marco Eilers and Peter Müller. 2018. Nagini: A Static Verifier for Python. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 596–603.

[13] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2 (apr 1985), 374–382. https://doi.org/10.1145/3149.214121

[14] Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. *Proc. ACM Program. Lang.* 7, ICFP (2023), 847–877. https://doi.org/10.1145/3607859

[15] Joseph Y. Halpern. 1987. A little knowledge goes a long way: simple knowledge-based derivations and correctness proofs for a family of protocols. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, British Columbia, Canada) *(PODC '87)*. Association for Computing Machinery, New York, NY, USA, 269–280. https://doi.org/10.1145/41840.41863

[16] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM* 60, 7 (June 2017), 83–92. https://doi.org/10.1145/3068608

[17] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295. https://doi.org/10.1109/32.588521

[18] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–55.

[19] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings.* 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

[21] L. Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* SE-3, 2 (1977), 125–143. https://doi.org/10.1109/TSE.1977.229904

[22] Leslie Lamport. 1992. Hybrid Systems in TLA$^+$. In *Hybrid Systems*, Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel (Eds.). Lecture Notes in Computer Science, Vol. 736. Springer, 77–102. https://doi.org/10.1007/3-540-57318-6_25

[23] Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 385–399. https://doi.org/10.1145/2837614.2837635

[24] Giuliano Losa and Mike Dodds. 2020. On the Formal Verification of the Stellar Consensus Protocol. In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020) (Open Access Series in Informatics (OASIcs), Vol. 84)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:9. https://doi.org/10.4230/OASIcs.FMBC.2020.9

[25] Kenneth L. McMillan and Oded Padon. 2020. Ivy: A Multi-modal Verification Tool for Distributed Algorithms. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II* (Los Angeles, CA, USA). Springer-Verlag, Berlin, Heidelberg, 190–202. https://doi.org/10.1007/978-3-030-53291-8_12

[26] J. Misra, K. M. Chandy, and Todd Smith. 1982. Proving safety and liveness of communicating processes with examples. In *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Canada) *(PODC '82)*. Association for Computing Machinery, New York, NY, USA, 201–208. https://doi.org/10.1145/800220.806698

[27] Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2017. Reducing liveness to safety in first-order logic. *Proc. ACM Program. Lang.* 2, POPL, Article 26 (dec 2017), 33 pages. https://doi.org/10.1145/3158114

[28] Longfei Qiu, Yoonseung Kim, Ji-Yong Shin, Jieung Kim, Wolf Honoré, and Zhong Shao. 2024. LiDO: Linearizable Byzantine Distributed Objects with Refinement-Based Liveness Proofs. *Proc. ACM Program. Lang.* 8, PLDI, Article 193 (jun 2024), 25 pages. https://doi.org/10.1145/3656423

[29] JOHANN SCHUMANN. 1995. Using the theorem prover SETHEO for verifying the development of a communication protocol in FOCUS -A Case Study-. Springer Nature, 338–352.

[30] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL (2018), 28:1–28:30. https://doi.org/10.1145/3158116

[31] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, Frans Kaashoek, and Nickolai Zeldovich. 2023. Grove: a Separation-Logic Library for Verifying Distributed Systems. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 113–129. https://doi.org/10.1145/3600006.3613172

[32] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David Basin. 2020. Igloo: Soundly Linking Compositional Refinement and Separation Logic for Distributed System Verification. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 152 (Nov. 2020), 31 pages. https://doi.org/10.1145/3428220

[33] N.V. Stenning. 1976. A data transfer protocol. *Computer Networks (1976)* 1, 2 (1976), 99–110. https://doi.org/10.1016/0376-5075(76)90015-5

[34] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2024. Trillium: Higher-Order Concurrent and Distributed Separation Logic for Intensional Refinement. *Proc. ACM Program. Lang.* POPL (2024). https://doi.org/10.1145/3632851

[35] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 357–368. https://doi.org/10.1145/2737924.2737958

[36] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2024. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. *Proc. ACM Program. Lang.* 8, POPL, Article 35 (jan 2024), 32 pages. https://doi.org/10.1145/3632877