# Distributed Protocol Combinators

Kristoffer Just Arndal Andersen[1] and Ilya Sergey[2]

[1] Aarhus University, Aarhus, Denmark
kja@cs.au.dk
[2] Yale-NUS College and NUS School of Computing, Singapore
ilya.sergey@yale-nus.edu.sg

**Abstract.** Distributed systems are hard to get right, model, test, debug, and teach. Their textbook definitions, typically given in a form of replicated state machines, are concise, yet prone to introducing programming errors if naïvely translated into runnable implementations.

In this work, we present *Distributed Protocol Combinators* (DPC), a declarative programming framework that aims to bridge the gap between specifications and runnable implementations of distributed systems, and facilitate their modeling, testing, and execution. DPC builds on the ideas from the state-of-the art logics for compositional systems verification. The contribution of DPC is a novel family of program-level primitives, which facilitates construction of larger distributed systems from smaller components, streamlining the usage of the most common asynchronous message-passing communication patterns, and providing machinery for testing and user-friendly dynamic verification of systems. This paper describes the main ideas behind the design of the framework and presents its implementation in Haskell. We introduce DPC through a series of characteristic examples and showcase it on a number of distributed protocols from the literature.

## 1 Introduction

Distributed fault-tolerant systems are at the heart of modern electronic services, spanning such aspects of our lives as healthcare, online commerce, transportation, entertainment and cloud-based applications. From engineering and reasoning perspectives, distributed systems are amongst the most complex pieces of software being developed nowadays. The complexity is not only due to the intricacy of the underlying protocols for multi-party interaction, which should be resilient to execution faults, packet loss and corruption, but also due to hard performance and availability requirements [2].

The issue of system correctness is traditionally addressed by employing a wide range of *whole-system* testing methodologies, with more recent advances in integrating techniques for formal verification into the system development process [5, 8, 20]. In an ongoing effort of developing a *verification* methodology enabling the *reuse* of formal proofs about distributed systems in the context of an open world, the DISEL logic, built on top of the Coq proof assistant [3], has been proposed as the first framework for mechanised verification of distributed systems, enabling modular proofs about protocol composition [24, 26].

The main construction of DISEL is a *distributed protocol* $\mathcal{P}$—an operationally described replicated *state-transition system* (STS), which captures the shape of the state of each node in the system, as well as what it *can* or *cannot* do at any moment, depending on its state. Even though a protocol $\mathcal{P}$ is not an executable program and cannot be immediately run, one can still use it as an *executable specification* of the system, in order to prove the system's intrinsic properties. For instance, reasoning at the level of a protocol, one can establish that a property $I : \mathsf{SystemState} \to \mathsf{Prop}$ is an inductive invariant *wrt.* a protocol $\mathcal{P}$.[3] A somewhat simplified main judgement of DISEL, $\mathcal{P} \vdash c$, asserts that an actual system implementation $c$ will *not* violate the operational specification of $\mathcal{P}$. Therefore, if this holds, one can infer that any execution of a program $c$, will not violate the property $I$, proved for protocol $\mathcal{P}$. DISEL also features a full-blown program logic, implemented as a Hoare Type Theory [19], which allows one to ascribe pre- and post-conditions to distributed programs, enforcing them via Coq's dependent types, at the expense of frequently requiring the user to write lengthy proof scripts.

While expressive enough to implement and verify, for instance, a crash-recovery service on top of a Two-Phase Commit [24], unfortunately, DISEL, as a systems *implementation* tool, is far from being user-friendly, and is not immediately applicable for rapid prototyping of composite distributed systems, their testing and debugging. Neither can one use it for teaching without assuming students' knowledge of Coq and Separation Logic [21]. Furthermore, system implementations in DISEL must be encoded in terms of low-level send/receive primitive, obscuring the high-level protocol design.

In this work, we give a practical spin to DISEL's main idea—disentangling protocol *specifications* from runnable, possibly highly optimised, systems *implementations*, making the following contributions:

- We distil a number of high-level distributed interaction patterns, which are common in practical system implementations, and capture them in a form of a novel family of *Distributed Protocol Combinators* (DPC)—a set of versatile higher-order programming primitives. DPC allow one to implement systems concisely, while still being able to benefit from protocol-based specifications for the sake of testing and specification-aware debugging.
- We implement DPC in Haskell, providing a set of specification and implementation primitives, parameterised by a monadic interface, which allow for multiple interpretations of protocol-oriented distributed implementations.
- We provide a rich toolset for testing, running, and visual debugging of systems implemented via DPC, allowing one to state and dynamically check the protocol invariants, as well as to trace their execution in a GUI.
- We showcase DPC on a variety of distributed systems, ranging from a simple RPC-based cloud calculator and its variations, to distributed locking [10], Two-Phase Commit [7], and Paxos consensus [12, 13].

[3] Examples of such properties include global-systems invariants, used, in particular, to reason about the whole system reaching a consensus [22, 25].

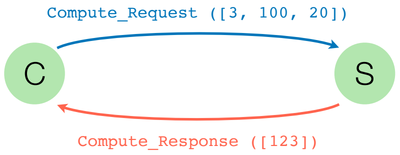
## 2 Specifying and Implementing Systems with DPC

In this work, we focus on message-passing asynchronous distributed systems, where each node maintains its internal state while interacting with others by means of sending and receiving messages. That is, the messages, which can be sent and received at any moment, with arbitrary delays, drops, and rearrangements, are the only medium of communication between the nodes. DPC takes the common approach of thinking of message-passing systems as shared-memory systems, in which each message in transit is allocated in a virtual shared "message soup", where it lingers until it is delivered to the recipient [24, 27].

The exact implementation of the *per-node* internal state might differ from one node to another, as it is virtually unobservable by other participants of the system. However, in order for the whole system to function correctly, it is required that each node's behaviour would be at least coherent with some notion of *abstract state*, which is used to describe the interaction protocol.

In the remainder of this section, we will build an intuition of designing a system "top-down". We will start from its specification in terms of a protocol that defines the abstract state and governs the message-passing discipline, going all the way down to the implementation that defines the state concretely and possibly combines several protocols together. For this, we use a standard example of a distributed calculator.

### 2.1 Describing Distributed Interaction



In a simple cloud calculator, a node takes one of two possible roles: of a *client* or of a *server*. A client may send a request along with data to be acted upon to the server (*e.g.*, a list of numbers `[3, 100, 20]` to compute the sum of), and the server in turn responds with the result of the computation, as shown on the diagram on the right. For uniformity of implementation, all messages, including the response of the server are lists of integers. Notice that this description does not restrict *e.g.*, the order in which a server must process incoming requests from the clients, which leaves a lot of room for potential optimisations on the implementation side.

In order to capture the behavioural contract describing the interaction between clients and servers, we need to be able to outlaw some unwelcome communication scenarios. For instance, in our examples, it would be out of protocol for the server to respond with a wrong answer (in general an issue of safety) or to the wrong client (in general an issue of security). A convenient way to restrict the communication rules between distributed parties is by introducing the *abstract state* describing specific "life stages" of a client and a server, as well as associated messages that trigger changes in this state—altogether forming an STS, a well-known way to abstractly describe and reason about distributed protocols [14, 15].

Let us now describe our calculator protocol as a collection of coordinated transition systems. The client's part in the protocol originates in a state `ClientInit` containing the input it is going to send to the server, as well as the server's

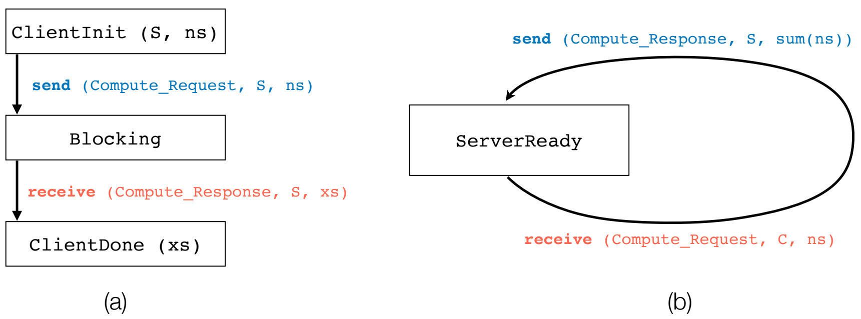

ClientInit (S, ns)
send (Compute_Request, S, ns)
Blocking
receive (Compute_Response, S, xs)
ClientDone (xs)
(a)
send (Compute_Response, S, sum(ns))
ServerReady
receive (Compute_Request, C, ns)
(b)

**Fig. 1.** State transitions for a client (a) and a server (b) in the calculator protocol.

identity. From this state, it can send a message to server `S` with the payload `[3, 100, 20]`. It then must wait, in a blocking state, for a response from the server.[4] Upon having received the message, the client proceeds to a third and final state, `ClientDone`. From here, no more transitions are possible, and the client's part in the protocol is completed. A schematic outline of the client protocol is depicted in Fig. 1 (a).

In our simplified scenario, the protocol for the server (Fig. 1, (b)) can be captured by just one state, `ServerReady`, so that receiving the request and responding to it with a correct result is observed as "atomic" by other parties, and hence, is denoted by a single composite transition. In other words, at the specification level, the server immediately reacts to the request by sending a response.

Notice that the protcol places no demands on the number of clients, servers or unrelated nodes in the network, nor does it restrict the number of instances of the protocol are running in a given network. The specification is ”local” to the parties involved (which in general can number arbitrarily many).

This "request/respond" communication pattern is so common in distributed programming that it is worth making explicit. We will refer to this pattern as a pure *remote procedure call* (RPC) and take it as our first combinator for protocol-based implementation of distributed systems.

### 2.2 Specifying the Protocol

We can capture the RPC-shaped communication in DPC by first enumerating all possible states of nodes in the protocol in a single data type. For the calculator, the states can be directly translated from the description above to the following Haskell data type:

```
data S = ClientInit NodeID [Int]
       | ClientDone [Int]
       | ServerReady
```

[4] Remember that this is a specification-level blocking, the implementation can actually do something useful in the same time, just not related to this protocol!

NodeID is a type synonym for Int, but any type with equality would serve. ClientInit contains the name of the server and the list to sum. ClientDone contains the response from the server. Next, we describe the only kind of exchange that takes place in a network of clients and servers communicating by following the RPC discipline. We do so by specifying when a client can produce a request in a protocol, and how the server computes the response. Perhaps, a bit surprisingly, no more information is needed, as the pattern dictates that clients await responses from servers, and the server responds immediately. This is the reason why need only enumerate two states for the client, eliding the one for blocking, as per Fig. 1 (a): the framework adds the third during execution by wrapping the states in a type with an additional Blocking constructor. [5] The following definition of compute outlines the specification of the protocol's STSs:

```
compute :: Alternative f ⇒ ([Int] → Int) → Protlet f S
compute f = RPC "compute" clientStep serverStep
  where
    clientStep s = case s of
      ClientInit server args → Just (server, args, ClientDone)
      _ → Nothing
    serverStep args s = case s of
      ServerReady → Just ([f args], ServerReady)
      _ → Nothing
```

As per its type, compute takes a client-provided function of type [Int] → Int, which is used by the server to perform calculations. The result of compute is of type Protlet f S, where S is the data type of our STS states defined just above and f is an instance encapsulating a possible non-determinism in a protocol specification. Later constructions will make integral use of non-determinism to, *e.g.*, decide on the next transition depending on the external inputs, and the parameter f serves to restrict what notion of non-determinism is used in the definition of protocols.[6] For now, the result of compute is entirely deterministic.

*Protlets* (*aka* "small protocols") are the main building blocks of our framework. A distributed protocol can be thought of as a family of protlets, each of which corresponds to a logically independent piece of functionality and can be captured by a fixed interaction pattern between nodes. In a system, each node can act according to one or more protlets, executing the logic corresponding to them sequentially, or in parallel. For this example, there is just the one exchange of messages, so a single protlet makes for the complete protocol description.

Our framework provides several constructors to build protlets from the data type description for the protocol state space and the operational semantics of its transitions. In the example above, RPC is a data constructor, which encodes the protlet logic by means of two functions. Its first argument, clientStep, prescribes that from ClientInit state, a node can send args to node server, and the response payload is later wrapped via ClientDone to form the succesor state. The second argument, serverStep, says that the state ServerReady can serve a

[5] See the discussion of executing specification in Section 3

[6] One can think of any protocol, whose diagram has a fork, as non-deterministic.

request in one step: receiving `args` and responding with `f args` in a singleton list, continuing in the same state. We have now completely captured the above intuitions and transition system of the calculator in less than ten lines of Haskell.

### 2.3 Executing the Specification

The immediate benefits of having an executable operational specification of a protocol is to be able to run it, locally and without needing full deployment across a network, ensuring that it satisfies basic sanity checks and more complex invariants.

The execution model for protlets is a small-step operational semantics, with the granularity of transitions being that of the involved protlets. We take as machine configurations the entire network of nodes and their abstract states.

In case several protlets of a similar shape are involved (*e.g.*, a node is involved in two or more RPCs), we distinguish them by introducing protlet labels, a solution that is standard for program logics for concurrency [4, 23]. Having introduced protlet labels, we can logically partition the local state of each node along the protlet instance space, maintaining a local state portion “per protlet”, per node. We represent this operational machine configuration as the datatype `SpecNetwork`, which is a record data structure maintaining an environment of protlets (indexed by their associated labels) and a protlet state for each node and protocol instance, so that the operational semantics changes one node’s one protlet’s state at a time. The following code creates a network for the calculator protocol with two nodes (identified by `0` and `1`), both running just one protlet (labelled with `0`), for the input for the example from Section 2.1:

```
addNetwork :: Alternative f ⇒ SpecNetwork f S
addNetwork = initializeNetwork nodeStates protocols
  where
    nodeStates = [ (0, [(0, ServerReady)])
                 , (1, [(0, ClientInit 0 [3, 100, 20])]) ]
    protocols = [(0, [compute sum])]
```

In any given network configuration, many actions can be possible. A node may be ready to initiate an RPC, or it might be ready to receive a message—many such actions may be enabled and relevant at once.[7] As the purpose of running the specification is to trace the possible behaviors in the protocol, we choose the next action to execute in the network by leaving the resolution to the *user* of the semantics. To do so, we implement the executable small-step relation as a monad-parameterised function capturing the possibility of non-determinism (hence `Alternative f`). This makes the implementation of the operational semantics simple, yet general, as it just needs to describe an `f`-ary choice or `f`-full collection of transitions at each step:

```
step :: (Monad f, Alternative f) ⇒ SpecNetwork f s → f (SpecNetwork f s)
```

The network can be “run” by iterating this small-step execution function with a suitable instance of `f`, a standard construction in implementation of a non-determinism in monadic interpreters.

[7] And their abundance is precisely why reasoning about distributed systems is hard.

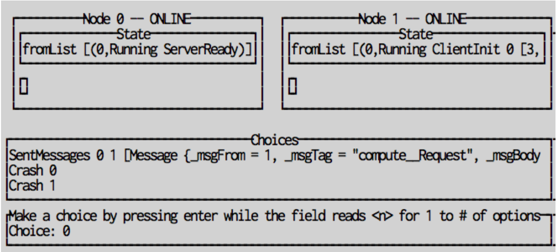
For example, we can instantiate the non-determinism to the classic choice of the list monad [17], which leads to enumerating every possible action. We can then iterate the function `step` by choosing a random possible transition, as in the following interaction with the library, where we explore the ”depth” of a single run of the protocol.

```
> length <$> simulateNetworkIO addNetwork
4
```

This is coherent with the first example we envisioned *wrt.* the protocol: there is (1) the initial state; (2) the state with the client awaiting response, but the message undelivered; (3) the state with the client waiting and the server having sent a response; and finally, (4) a terminal state with the client done.

The non-determinism can be similarly resolved by enumerating all possible paths through a protocol, up to a certain trace length if the execution space is not finite. If the state space of a network is finite, this can yield actual finite-space model checking procedures. In the following subsection, we will explore another alternative to resolving the non-determinism, yielding an unusual yet very useful execution method.

### 2.4 Interactive Exploration with GUI



```
Node 0 — ONLINE                            Node 1 — ONLINE
State                                      State
fromList [(0,Running ServerReady)]         fromList [(0,Running ClientInit 0 [3,
[]                                         []

Choices
SentMessages 0 1 [Message {_msgFrom = 1, _msgTag = "compute__Request", _msgBody
Crash 0
Crash 1

Make a choice by pressing enter while the field reads <n> for 1 to # of options
Choice: 0
```

**Fig. 2.** The interactive exploration tool, loaded with the calculator protocol.

By delegating the decision of which transition to follow to the user of an application that performs this simulation, we can allow the client of the framework to explore the network behaviour interactively. The DPC library provides a command-line GUI application facilitating interactive exploration of distributed networks step-by-step. Provided an initial network specification like the one described previously, one can start the session by typing the following:

```
> runGUI addNetwork
```

This yields the interface displayed in Fig. 2. By choosing specific transitions in sequence, the user can evolve and inspect the network at each step of execution. This is useful for protocol design and debugging, and can help understand the dynamics of a protocol, and the kinds of communication patterns it describes.

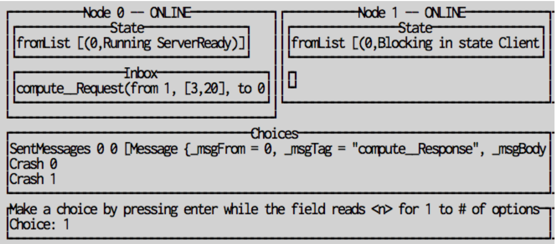
For example, in Fig. 3 we show the subsequent prompt after showing the selection of Option 1:

```
SentMessages 0 1 [Message {_msgFrom = 1, _msgTag = "compute__Request",...
```

`SentMessages` is a human readable piece of data that represents the option of sending in protocol instance `0`, from node `1` the message with sender `1` of tag `"compute__Request"`. Here, the recipient and message content is elided for issues of screenspace, but as the window is enlarged, so is the depth of information provided to the client of the framework.

The state view is then shows that Node `0` now has said message waiting for it in the soup, and Node `1` is now blocking. The user is then presented with subsequent possible choices, here the option for the calculator to receive the request and send the response in one atomic action, as dictated by the protocol.

Additionally, as can be seen in Fig. 2, in the interactive tool we enrich the possible transitions at every step with the possibility of a node to go off-line. In effect, it means it will stop processing messages, modelling a benign (non-byzantine) fault. Other nodes cannot observe this and will "perceive" the node as not responding. This, however, becomes very useful when we move to explore protocols that allow for partial responses among a collection of nodes, as in the case of crash-resilient consensus protocols.



```
fromList [(0,Running ServerReady)]
compute__Request(from 1, [3,20], to 0
fromList [(0,Blocking in state Client
[]
SentMessages 0 0 [Message {_msgFrom = 0, _msgTag = "compute__Response", _msgBody
Crash 0
Crash 1
Make a choice by pressing enter while the field reads <n> for 1 to # of options
Choice: 1
```

**Fig. 3.** Choosing option `1` in the prompt from Fig. 2.

### 2.5 Protocol-Aware Distributed Implementations

Distributed systems protocols serve as key components of some of the largest software systems in use. The actions taken in the protocol are governed by programs outside the key protocol primitives, so it is vital that implementations can integrate with software components in real general-purpose languages. We here present such a language with primitives for sending and receiving messages as an embedded domain-specific language (EDSL) in Haskell. This allows use of the entire Haskell toolkit in engineering efficient optimised implementations relying on distributed interaction.

Naturally, as implementations deviate from the protocols (in the way they, *e.g.*, implement internal state), we want to ensure that the they still adhere to the protocol as specified. To achieve this, we introduce primitives for *annotating* implementations with protocol-specific assertions. These annotations can be ignored by execution-oriented interpretations aiming for efficiency rather than verification guarantees.

The following code implements a calculator server in plain Haskell using `do`-notation to sequence effectful computations. The effects are described by type class constraints: `MessagePassing` provides a `send` and `receive` primitive, and `ProtletAnnotations` providing the `enactingServer` primitive, explained below.

```
addServer :: (ProtletAnnotations S m, MessagePassing m) ⇒ Label → m a
addServer label = loop
 where
   loop = do
     enactingServer (compute sum) $ do
       Message client _ args _ _ ← spinReceive [(label, "Compute__Request")]
       send client label "Compute__Response" [sum args]
     loop
```

By using type classes describing operations, we allow for several different interpretations of this code. For instance, by interpreting the `send` and `receive` as POSIX Socket operations, we obtain a subroutine in the IO Monad, Haskell's effectful fragment, that we can integrate into any larger development with no interpretive overhead. The `spinReceive` operation is defined using recursion and a primitive `receive` operation that attempts to receive an incoming message with a tag from amongst a list of canidate message tags in a non-blocking manner.

The body of `addServer` is annotated with a `(compute sum)` protlet, enforcing that the server responds to the client atomically (in terms of message passing) and to perform the `sum` function (or something observationally equivalent) on the supplied arguments. By bracketing the `receive` and `send` in the `enactingServer` primitive, the implementation declares its intent to conform to the server role of the RPC, as dictated by the protocol. Once we have a client to play the other role in the protocol, we will demonstrate how this intent can be checked dynamically. The message tags that appear in the code are *by convention* the tags used in the RPC protocol, *i.e.*, the name of the protocol with a suffix indicating the role in the RPC that the message plays.

In contrast DISEL and other static verification frameworks that enforce protocol adherence via (dependent) type systems (embedded in Coq or other proof assistants) [11,24], we verify protocol properties dynamically. The tradeoff is that of coverage versus annotation and proof overhead. We can, through exploiting executable specifications, check that *a single run of a program* adheres to a protocol. Notice that `addServer` is, like the specification of the `compute` protlet, agnostic in the number and kinds of other nodes in the network. Its behaviour is locally and completely described by its implementation, and is segregated from interfering with unrelated protlets via the `label` parameter. We refer the reader to the development for a number of client component implementations.

Let us now reap the benefits of protocol-aware distributed programming enabled by DPC and *dynamically check* that the implementations do indeed follow the abstract protocols. We achieve this by interpreting the EDSL into a datatype of abstract syntax trees (`AST`) that makes it possible to inspect their evaluations at run time. We give a small-step structural operational semantics to this language, and, precisely like the exectuable specifications, lift the evaluation of a single program to that of an entire network of programs, by assigning each program a node identifier in the network. Here, the global state (of type `ImplNetwork m Int`, with `m` constrained as in `addServer`/`addClient`) is just the message soup, and the node-local state is the program itself. Such an evaluation is implemented by the following function.

```
runPure :: ImplNetwork (AST s) a → [(TraceAction s, ImplNetwork (AST s) a)]
```

Here, the `AST` data type is the HOAS AST for message-passing implementations to be interpreted. The result of running the network is a (possibly infinite) list of `TraceAction`s and the network configurations they lead to. We can simulate a full run of the network by taking the last network in this list, provided the network terminates. Messages can be examined by considering the soup component at every step of evaluation.

We can verify that our implementation indeed adheres to the desired protocol by the trace produced by `runPure` on a network configuration, ensuring that (a) every observable action is compatible with the state that the node is supposed to be in, and (b) checking the messages expected from these states. For this, we implement yet another operational semantics, where the machine configuration is a protocol state for every node id, and the program is a trace of primitive actions. The interpreter faults if the current action is not applicable to the state, or sends or receives messages not prescribed by the specification. We can run the adherence checker on a *prefix* (*e.g.*, of length 15) of the infinite trace as follows:

```
> checkTrace addNetwork $ fmap fst . take 15 $ runPure addConf
Right ()
```

The result of `Right ()` indicates success: the trace did indeed conform to the protlet annotations of the program, assuming the initial state of the implementations in `addConf` assumed an initial abstract state corresponding to the the network state of `addNetwork`.

What happens if we introduce a mistake in the implementation? For instance, if we erroneously annotate the server as intending to serve a `product` function (instead of `sum`), we will fail protocol adherence, because the specification does not agree on the content of the messages. In a different scenario, if we run the client implementation *twice*, the checker would report an error, as this is not allowed by the protocol: the client would have brought itself to the terminal state `ClientDone` by the first RPC, and, hence, cannot proceed. By enriching dynamic testing with protocol adherence checks we believe we can achieve greater assurances of the correctness of our implementations without resorting to use full-blown verification frameworks [8, 24].

## 3 Framework Internals

### 3.1 The Specification Language

A full distributed system specification consists of a collection of nodes, each assigned a unique node identifier, and a collection of protlets for each instance label. A node owns local state, partitioned according to protocol instance labels. A protlet describes one exchange pattern between parties. A collection of protlets over the same state space then describe an entire protocol.

In the overview we saw the simplest protlet, the pure RPC, but through exploration of examples and case studies, we have discovered a number of such patterns, each more general than the previous. These are implemented as extensions to the `Protlet` data type. One such is the broadcast protlet, integral for describing multi-party protocols.[8]

```
data Protlet f s =
  | RPC          String (ClientStep s) (ServerStep s)
  | Broadcast    String (Broadcast s)  (Receive s) (Send f s)
  | ...
```

[8] We elide the other protlet constructors, which can be found in our implementation.

The component functions of the protlets reuse a number of common type abbreviations, here `ClientStep`, `Send` etc. All are at work in the above listing. This common structure unifies their implementation in the operational semantics. The expansion of, *e.g.*, the `Broadcast` synonym is as follows:

```
type Broadcast s = s → Maybe ([(NodeID, [Int])], [(NodeID, [Int])] → s)
```

This models a "partial" function on states `s`, saying under which conditions a node can initiate a broadcast, by enumerating the recipients and the body of the messages to them, along with a continuation processing the received answers with their associated senders. This continuation is stored in the implicit blocking state during actual execution of the specification.

The specification language is given a non-deterministic operational semantics as described in Section 2.3. Recall the network step function:

```
step :: (Monad f, Alternative f) ⇒ SpecNetwork f s → f (SpecNetwork f s)
```

It is implemented by computing an `f`-full of possible transitions for every node in the network and combining the result of taking all possible transitions on the current network. The key operation of `step` is a dispatch on the current protocol state of a node:

```
  case state of
    BlockingOn _ tag f nodeIDs k →
      resolveBlock label tag f nodeID inbox nodeIDs k
    Running s → do
      protlet ← fst <$> oneOf (_globalState Map.! label)
      stepProtlet nodeID s inbox label protlet
```

The constructors `BlockingOn` and `Running` are supplied by the framework. The first is used to track the terms under which a node is blocking: what message(s) it needs to continue and from whom. `resolveBlock` computes whether the conditions are met for the current node to continue.

Here, `_globalState` is the mapping of collections of protlets (*i.e.*, a protocol) from instance labels. We then choose between protlets using `oneOf :: [a]→f a`. `stepProtlet` dispatches control based on a case distinction on the protlet constructor: for example, here is the branch for the `Broadcast` protlet:

```
stepProtlet :: (Monad m, Alternative m) ⇒
  NodeID → s → [Message] → Label → Protlet m s →  m (Transition s)
stepProtlet nodeID state inbox label protlet = case protlet of
  ...
  Broadcast name broadcast receive respond →
    tryBroadcast label name broadcast nodeID state inbox <|>                -- (1)
    tryReceive label (name ++ "__Broadcast") receive nodeID state inbox <|>      -- (2)
    trySend label respond nodeID state inbox                                -- (3)
  ...
```

A node attempting to advance a protocol using the `Broadcast` protlet can do so if it is (1) a client ready to perform a broadcast; (2) a server ready to receive such a broadcast; or (3) a server that is ready to respond to a broadcast. The `try` functions all follow the same structure: check that the user-provided protlet component functions apply, and if so, generate an appropriate transition. For instance, here is the signature of one such function for `Broadcast`:

```
tryBroadcast :: Alternative f ⇒ Label → String → Broadcast s →
                NodeID → s → [Message] → f (Transition s)
```

*Interpretations of Protocols.* As described in Section 2.3, the operational semantics of protocols can be instantiated to obtain different interpretations. We here look at bounded model checking mentioned in passing in the overview. We can use the `List` monad to enumerate all execution paths in a breadth-first manner:

```
simulateNetworkTraces :: SpecNetwork [] s → [[SpecNetwork [] s]]
```

This yields a list-of-lists where the $n$th list contains all possible states after $n$ steps of execution, in a breadth first enumeration of the state space. Each constituent list of states is necessarily finite, but the list-of-lists need not be in the case of infinite network executions. By virtue of Haskell's lazy evaluation, such a computational object is useful. We can then write a procedure that, given a trace, applies a boolean predicate at every step of the trace.

```
checkTrace :: Invariant m s Bool → m → [SpecNetwork f s] → Either Int ()
```

The `Invariant` data type is an abbreviation for a boolean predicate on the type `s` that additionally takes some "meta-data" `m`, like "roles" in a protocol, needed to express the invariant. The procedure `checkTrace` returns `Right ()` to signify that there were no violations of the invariant, while it returns `Left n` to report that the `n`th state was the first state to violate the invariant. With this language of predicates we can build invariants and with the aforementioned checking procedure we can perform (bounded) checking that an invariant is in fact inductive (*i.e.*, holds for each state). In the case of a finite state space, this amounts to real verification of inductive invariants. The most sophisticated example we have successfully specified is an inductive invariant for a Two-Phased Commit protocol [24], for which we refer the curious reader to the implementation.

### 3.2 The Implementation Language

The monadic langugage for message-passing programs is implemented as an EDSL in Haskell. This has the benefit of providing all the standard tools for writing Haskell programs; all the abstraction mechanisms and organisational principles are at hand to write sophisticated software, including lazy evaluation, higher-order functions, algebraic data types and more. By virtue of the modularity offered by the approach of EDSLs, it is straightforward to give multiple interpretations of such programs.

At the time of this writing DPC's implementation fragment came with three interpretations of the monadic interface:

1. The `AST` monad used for dynamic verification of implementation adherence of the implementations to protocols, and covered in detail in Section 2.5.
2. A shared-memory based interpretation where nodes are represented as threads, and message passing is performed by writing to shared message queues using non-blocking concurrency primitives.
3. An interpretation for distributed message passing.

In the third case (true distribution), we give an interpretation into `IO` computations performing message passing through POSIX Sockets. For this, each computation needs an "address book" mapping `NodeIDs` to physical addresses (concretely, IP adresses and ports). Additionally, each program will have access to a local mailbox, represented by a message buffer being filled by a local thread whose only function is to listen for messages. These two pieces of data are collected in a record of type `NetworkContext`. Computations running in such a context are captured in a type synonym over the `ReaderT` monad transformer:

```
newtype SocketRunnerT m a = SocketRunnerT {
    runSocketRunnerT :: ReaderT NetworkContext m a }
```

What follows is the implementation of the `send` primitive in this particular instance of the message-passing interface:

```
instance (MonadIO m) ⇒ MessagePassing (SocketRunnerT m) where
  send to lbl tag body = do
    thisID ← this
    let p = encode $ Message thisID tag body to lbl
    peerSocket ← (!to) <$> view addressBook
    void . liftIO $ Socket.send peerSocket p mempty
```

The code for sending messages is, thus, implemented in a form of a `Reader`-like computation over an `IO`-capable monad `m` as indicated by the `MonadIO` constraint. It starts by building a `Message` containing the supplied tag, body, receiver (`to`) and label, along with the executing nodes ID, as supplied by another primitive, `this`. It then uses `encode` to serialize this message into bytestring `p`. `p` is then sent to the appropriate `peerSocket`, as resolved by the `addressBook`, using the `System.Socket.Send` operation from the POSIX Socket library for Haskell. The monadic glue code (and the rest of the Haskell toolkit) is interpreted by choosing an appropriate base monad for the interpretation, *e.g.*, the `IO` monad. Ultimately, we build the following function for running the system:

```
defaultMain :: NetworkDescription → NodeID → SocketRunner a → IO ()
```

It takes a `NetworkDescription`, which maps `NodeIDs` to physical addresses, a `NodeID` with which to identify this node, and a computation in the above described interpretation of message passing programs. The result is an `IO ()` computation that establishes (if run on each machine) a fully connected mesh network with every node in the supplied network description, and then proceeds to run the supplied computation, passing messages accordingly. This interpretation can be used to facilitate integration of DPC-based implementations with real Haskell code once they have been assured to comply with their protocols.

## 4 Evaluation

The implementation of DPC is publicly available online for extensions and experimentation.[9] We now report on our experience of using DPC for implementing and validating some commonly used distributed systems.

[9] https://github.com/kandersen/dpc

| Protocol | Impl | Protlets | LOC | RPC | ARPC | Notif | Broad | OneOf | Quorum |
|---|---|---|---|---|---|---|---|---|---|
| Calculator | ✓ | 1 | 10 | ✓ | ✓ |  |  | ✓ |  |
| Lock Server |  | 4 | 73 | ✓ | ✓ | ✓ |  |  |  |
| Concurrent Database |  | 3 | 23 | ✓ |  |  |  |  |  |
| Two-Phase Commit |  | 2 | 43 |  |  |  | ✓ |  |  |
| Paxos | ✓ | 2 | 42 |  |  |  |  |  | ✓ |

**Table 1.** A summary for implemented systems: protocol, runnable implementation, count of constituent protlets, size of encoding (lines of code), employed combinators.

### 4.1 More Examples

In order to evaluate the framework, we have encoded a number of textbook distributed protocols, translating their specifications to the abstractions of DPC. By doing so, we were aiming to answer the following research questions:

1. Are our `Protlet`-based combinator sufficiently expressive to capture a variety of distributed systems from the standard literature in a natural way?
2. Is it common to have realistic protocols that require *more than one* combinator, *i.e.*, can be efficiently decomposed into multiple `Protlets`?
3. What is the implementation burden for encoding systems using DPC?

The statistics for our experiments is summarised in Table 1.

The framework has been shaped by the explorations of protocols that we have made, but we believe that the answer to Q1 is affirmative, supported by the variety of protocols we have so far explored. The answer to Q2 is also affirmative. Complex protocols from literature decompose into interactions shaped as RPCs, notifications, *etc*, and we manage to capture all of them in protlets. Simply put, for every arrow in a diagram of the network indicating a communication channel, the protocol has a protlet detailing the exchanges occuring across that channel. For instance the two-phase protocols like Paxos and Two-Phase Commit (2PC) naturally decompose into two broadcast/quorum phases, while more asymmetric protocols like distributed locking [10] requires as many as four protlets.

Regarding Q3, the lines of code versus complexity of protocol are indicative of a positive relationship between complexity and effort to encode a protocol, which is desireable. That is, a lot of complexity is encapsulated by the treatment of combinators, so the coding effort in the framework is very light.

The nature of the verification that the framework enables is naturally not strictly sound (as it is dynamic), but techniques like bounded model checking are readily explorable. With it, we have been able to validate, *e.g.*, correctness for the 2PC protocol [24], a not an insignificant proof burden.

The framework also affords exploration in other directions than we have mentioned so far. We have experimented with enriching the message passing language with operations for shared-memory concurrency and thread-based parallelism. The database example in the table uses *node-local* threads to maintain a database that is served by two different threads. Our approach to dynamic checking of protocol adherance scales to concurrency, and we have a concurrent Calculator server serving *multiple* arithmetic functions *in parallel*.

### 4.2 A Case Study: Constructing and Running Paxos Consensus

For a representative exploration of the capabilities of DPC we turn to a study of the Paxos Consensus [2,6,12]. Paxos solves a problem of reaching a consensus on a single value agreed upon across multiple nodes, of which a subset acts as proposers (who suggest the values) and another, complementary subset acts as acceptors (who reach an agreement). The nature of the Paxos algorithm lends itself well to interactive exploration and the specification should be robust to issues that appear specifically in distributed systems, like arbitrary interleaving of messages, message reorderings, and nodes going offline. The tools we have developed so far are enough to explore these aspects of the protocol.

We can specify this protocol in DPC with relatively little code. We further generalise the `Broadcast` combinator to "quorums" — broadcasts that await only a certain number of responses before proceeding. We introduce another entry in our `Protlet` datatype for capturing this pattern.

```
data Protlet f s = ...
  | Quorum String Rational (Broadcast s) (Receive s) (Send f s)
```

The `Quorum` protlet is and acts identical to the Broadcast protlet, but it is further instrumented by a rational number indicating the number of responses to await before proceeding. We encode the dissection of nodes into proposers and acceptors directly in the state of the protocol, similar to how we dissected the state space of the cloud server along Client/Server lines. The proposer starts in `(ProposerInit b v as)` with the desire to propose to acceptors as the value `v` with priority (*ballot*) `b`. We encode this with a quorom protlet:

```
prepare :: Alternative f ⇒ Label → Int → Protlet f PState
prepare label n = Quorum "prepare" ((fromIntegral n % 2) + 1) propositionCast ...
  where
    propositionCast = λcase
      ProposerInit b v as → Just (zip as (repeat [b]), propositionReceive b v as)
      _ → Nothing
```

Here, `prepare` is parameterised by the number of participants. Hence, the protlet dictates we should wait for a majority quorum, to avoid ties in the system. The listing shows the initiation of the first broadcast as representative of the rest of the implementation. The proposer starts in an `ProposerInit` state, in which it initiates a broadcast poll of all `as` acceptors, sending its ballot `b`.

The second phase of the protocol is encoded as another `Quorum` protlet, where the proposers react to the outcome of the responses on the first polling. The interactive exploration tool can be used to explore, for instance, the robustness of the protocol with respect to crashing participants versus crashing proposers, and why a quorum size of $\left(\frac{n}{2} + 1\right)$ acceptors is sufficient for reaching consensus.

The explored implementation demonstrates use of the *state* monad to organise the acceptor as an effectful program, and a *callback* to provide the ballot to the proposer, using features of Haskell, while retaining the benefits of the framework. Neither effect is possible to express at the protocol specification level.

## 5 Related Work

*Declarative programming for distributed systems.* In the past five years, several works were published proposing mechanised formalisms for scalable verification of distributed protocols, both in synchronous [5] and asynchronous setting [24, 27]. All those verification frameworks allow for executable implementations, yet the encoding overhead is prohibitively high, and no abstractions for specific interaction patterns are provided in any of them. Most of the DSLs for distributed systems we are aware of are implemented by means of extracting code rather than by means of a shallow DSL embedding [9, 16, 18]. MACE [9], a C++ language extension and source-to-source compiler, provides a suite of tools for generating and model checking distributed systems. DISTALGO [18] and SPLAY [16] extract implementations from protocol descriptions.

In a recent work, Brady has described a discipline of protocol-aware programming in IDRIS [1], in which adherence of an implementation to a protocol is ensured by the host language's dependent type system, similarly to DISEL, but in a more lightweight form. That approach provides strong static safety guarantees; however, it does not provide dedicated combinators for specific protocol patterns, *e.g.*, broadcasts or quorums.

DPC's protlets adapt DISEL's protocols, that are phrased exclusively in terms of *low-level* send/receive commands, which should be instrumented with protocol-specific logic for each new construction. While it is possible to derive DPC's protlets in DISEL, extracting them and ascribing them suitable types requires large annotation overhead.

## 6 Conclusion and Future Work

Declarative programming over distributed protocols is possible and, we believe, can lead to new insights, such as better understanding on how to structure systems implementations. Even though there are several known limitations to the design of DPC (for instance, in order to define new combinators, one needs to extend `Protlet`), we consider our approach beneficial and illuminating for the purposes of prototyping, exploration, and teaching distributed system design.

In the future, we are going to explore the opportunities, opened by DPC, for randomised protocol testing and lightweight verification with refinement types.

*Acknowledgements.* We thank PADL'19 referees for their many helpful suggestions. The authors' work on this project has been supported by the grant by UK Research Institute in Verified Trustworthy Software Systems (VeTSS).

## References

1. E. Brady. Type-driven development of concurrent communicating systems. *Computer Science (AGH)*, 18(3), 2017.
2. T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, pages 398–407. ACM, 2007.
3. Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2018.
4. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.

5. C. Dragoi, T. A. Henzinger, and D. Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415. ACM, 2016.
6. Á. García-Pérez, A. Gotsman, Y. Meshman, and I. Sergey. Paxos Consensus, Deconstructed and Abstracted. In *ESOP*, volume 10801 of *LNCS*, pages 912–939. Springer, 2018.
7. J. N. Gray. Notes on data base operating systems. In *In Operating Systems*, pages 393–481. Springer, 1978.
8. C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. IronFleet: proving practical distributed systems correct. In *SOSP*, pages 1–17. ACM, 2015.
9. C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *PLDI*, pages 179–188. ACM, 2007.
10. M. Kleppmann. How to do distributed locking. https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html, 08 Feb 2016.
11. M. Krogh-Jespersen, A. Timani, M. E. Ohlenbusch, and L. Birkedal. Aneris: A logic for node-local, modular reasoning of distributed systems, 2018.
12. L. Lamport. The Part-Time Parliament. *ACM TOPLAS*, 16(2):133–169, 1998.
13. L. Lamport. Paxos made simple, 2001.
14. L. Lamport and F. B. Schneider. Formal foundation for specification and verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, volume 190 of *LNCS*, pages 203–285. Springer, 1985.
15. B. W. Lampson. How to build a highly available system using consensus. In *WDAG*, 1996.
16. L. Leonini, E. Riviere, and P. Felber. SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *NSDI*, pages 185–198. USENIX Association, 2009.
17. S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In *POPL*, pages 333–343. ACM Press, 1995.
18. Y. A. Liu, S. D. Stoller, B. Lin, and M. Gorbovitski. From clarity to efficiency for distributed algorithms. In *OOPSLA*, pages 395–410. ACM, 2012.
19. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240, 2008.
20. C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon web services uses formal methods. *Commun. ACM*, 58(4), 2015.
21. P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*. Springer, 2001.
22. G. Pîrlea and I. Sergey. Mechanising blockchain consensus. In *CPP*, pages 78–90. ACM, 2018.
23. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM, 2015.
24. I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols. *PACMPL*, 2(POPL):28:1–28:30, 2018.
25. R. van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Comp. Surv.*, 47(3):42:1–42:36, 2015.
26. J. R. Wilcox, I. Sergey, and Z. Tatlock. Programming Language Abstractions for Modularly Verified Distributed Systems. In *SNAPL*, pages 19:1–19:12, 2017.
27. J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368. ACM, 2015.