Reowolf 1.0

Project Documentation

Christopher A. Esterhuyse

Hans-Dieter A. Hiep

Centrum Wiskunde & Informatica

November, 2020

Contents

Chapter 1

Introduction

This document serves as the documentation and specification of the Reowolf project, aiming to provide connectors as a generalization of BSD-sockets for multi-party communication over the Internet.

The Reowolf project started in November, 2019. It is supported by the Next Generation Internet Privacy & Trust Fund (NGI-Zero PET), a fund established by NLnet with financial support from the European Commission's Next Generation Internet programme, under the aegis of DG Communications Networks, Content and Technology under grant agreement No. 825310 (Horizon 2020). NLnet Foundation is a non-governmental public benefits organization that "has been financially supporting organizations and people that contribute to an open information society. It funds those with ideas to fix the internet."

1.1 Audience

Readers are expected to be somewhat familiar with Internet architecture and basic protocols such as IP, ICMP, TCP, UDP; existing routing infrastructure and services such as DHCP, DNS; systems programming using the C language with POSIX or UNIX-like operating systems; network programming experience using sockets and application layer protocols such as HTTP, FTP, SMTP, or any other application layer protocol typically implemented using sockets; and experience using a programming language similar to C or Java.

Chapters whose contents build upon existing literature include a final section on bibliographical remarks, giving readers an entry point to understanding more technical details for those missing relevant background knowledge.

1.2 Vision and Mission

As more Internet traffic is encrypted to enhance the privacy of its users, its nature is less insightful to network operators. This might lead to inefficient routing of traffic, the inability to monitor for abuse, and unfair networking practices. The vision of this project is to build new communication methods that increase user privacy and trustworthiness of Internet infrastructure, aimed at separating private message contents from the description of network communication. In the end, this might result in an alternative to deep packet inspection for network operators that better

protects privacy for users, and higher performance of complex data streaming applications by increasing throughput and decreasing latency.

Reowolf aims to replace BSD sockets with *connectors* for communication on the Internet. Connectors generalize sockets, facilitating multi-party communication according to an explicitly-expressed *protocol description* in Reowolf's Protocol Description Language (PDL). The use of these textual protocol descriptions allows applications to express their requirements on the nature of the traffic to the underlying network communication infrastructure at a higher level of abstraction. This abstract representation serves a dual purpose: (1) applications are able to relegate the implementation of coordination logic to Reowolf itself, becoming more modular and re-usable, and (2) protocol descriptions act as a vehicle for communicating the abstract requirements and intent of communication sessions to humans and machines alike, allowing for optimizations in middleware all along the network path, and the development of standard libraries for protocols with useful, verified properties.

From a more practical point of view, Reowolf allows the implementation of application-specific firewalls. In current practice, it is unknown to operating systems what network behavior applications do expect and wish to handle normally, versus what network behavior is unwanted or unsolicited. By using Reowolf, applications declare using the Protocol Description Language all permitted interactions over one or more data streams; other behavior is explicitly disallowed. Thus, traffic that does not conform to the protocol description is necessarily unwanted, and Reowolf implementations can use this information for increasing visibility in protocol violations that may signal network intrusion or defects. Moreover, this inspection of traffic can happen also within the network infrastructure, not only on the edge of the network. If applications apply this technique, a more reliable network infrastructure in the long run may result.

1.3 Goals

The Reowolf project's main objective is the design and implementation of connectors. We first focus on functional aspects to establish a baseline for correctness and functionality. Specifically, our core goals are:

1. designing the connector application programming interface (Chapter 3),
2. designing the protocol description language (Chapter 4),
3. defining and relating the semantics of protocols and connectors (Chapter 5),
4. designing and implementing a reference implementation, (Chapter 6),
5. demonstrating functionality through a number of detailed examples, (Chapter 10)

After we are effectively able to demonstrate functionality, we will focus on reliability, efficiency, and backwards compatibility. What follows explores various avenues of performance optimization and features for increased safety to demonstrate Reowolf's potential. More specifically, further goals are:

1. demonstrating prevention, detection and recovery from distributed peers deviating from the configured protocol (Chapter 7)
2. demonstrating a number of optimization opportunities, possibly applicable to only a subset of the full protocol description language. (Chapter 8),

3. maintain backwards compatibility with existing Internet infrastructure, and existing socket-based applications. (Chapter 9).

This project aims to produce a high-quality reference implementation to form the groundwork for future research assignments and development efforts.

1.4 Milestones

This section presents an overview of the document as a collection of incremental deliverables, categorized by the appropriate milestone.

	Milestone	Due Date	Document Deliverable
1.	Design of the API and protocol language	02/12/2019	Chapter 2 provides an overview on the design of connectors and protocols, and their relationship to existing socket programming paradigms. Chapter 3 details API and usage of connectors for communication as presented to an application developer. Chapter 4 defines Reowolf's Protocol Description Language (PDL), whose role as a semantics for communicating connectors is described in Chapter 5.
2.	Naïve implementation	03/02/2020	Chapter 6 provides a textual description for the reference implementation for creation, management and usage of connectors for communication and explains the workings of the implementation. This includes sections for elaborating the means by which connectors identify and execute communication to the satisfaction of all peers' protocols (Section 6.1), and distributed consensus and synchronization to maintain a consistent distributed state. Chapter 10 demonstrates, through concrete examples in the C language, the use of the implementation for network communication.
3.	Interoperability	03/08/2020	Chapter 9 explores extensions to the reference implementation to facilitate interoperability with applications built atop BSD-style network sockets. It explains the so-called 'socket sandwich'. Connectors provide UDP mediator components for backwards-compatible communications over UDP with hosts on the network. This chapter describes the implementation of backwards compatibility, and provides a concrete example of applications built on top of pseudo-sockets in the C language.

4.	Deviation detection	30/03/2020	Chapter 7 expands upon the reference implementation from Chapter 6 to make connectors robust to the deviations of individual networked peers from the configured protocol. This includes detection and removal of preventable deviations, and a roll-back mechanism for recovering a consistent state when conflicts between peers' protocols cannot be resolved within an arbitrary timeout.
5.	Local optimization	03/04/2020	Chapter 8 describes improvements to the reference implementation which optimizes the utilization of resources supporting communicating components within a shared memory space. This includes the decoupling of logical and transport-layer channels, and allows for more efficient transport and sharing of message contents between components.
6.	Middleware optimization	29/06/2020	Chapter 8 describes modifications and additions to the reference implementation which facilitates the detection and application of optimizations that arise in the context of particular communication sessions. Ultimately, these take the form of processes coordinating efforts to share session information, and mutating their internals to simplify and transform the nature of the session in a fashion that has no observable effect on user programs.
7.	Benchmarking	05/10/2020	Chapter 11 shows performance benchmarking of the connector runtime implementation. This includes (a) a description of the experimental design and methodology, with concrete code snippets, (b) concrete measurements presented in plots, (c) an objective interpretation of the measurements, and (d) discussion of the relationship between observations, and properties of the implementation. The set of tests are incremental in nature, teasing out properties in isolation first for simple connectors on a single host, and then ranging up to large, complex connectors spanning the internet. Finally, the effects of session optimizations described in Chapter 8 are exemplified, comparing session performance before and after session transformations.

8. Documentation	02/11/2020	The documentation is revised to provide a consistent view of the final design and implementation, with inconsistent, preliminary information retained in the appendix to avoid confusion, while preserving relevant, historical work. Chapter 10 gives an entry-point for users, in the form of a bottom-up guide to network programming with the connector API, and protocol programming with PDL. The section provides concrete examples for essential protocols, expressed in PDL, exemplifies their usage, and reasons about their behavior in the abstract. The chapter also shows the correspondence between PDL and its predecessor language, Reo, by comparing and contrasting the two languages, and embedding canonical Reo protocols in PDL. Finally, example programs demonstrate the ways in which connectors are able to inter-operate with sockets.

Chapter 2

Design Overview

This chapter serves as a top-down overview of the project's most essential concepts, and ubiquitous terminology. The chapter introduces, and inter-relates concepts to be expanded upon in other chapters in a bottom-up fashion.

2.1 Terminology

Reowolf sits at the union of the domains of discourse of computer networking, coordination languages, and formal logic. By design, the conventions set by BSD-style sockets are followed. Reowolf uses familiar terminology for concepts already present in each of its domains. However, certain terms have a different meaning in each domain. To avoid such ambiguity, this section enumerates the most essential terms and, for each, gives a short definition.

- **Protocol**
    1. *Physical protocol*, protocols that enable electromagnetic signal exchange, for example Fast Ethernet, Gigabit Ethernet and Wi-Fi.
    2. *Internet protocol*, protocols that enable exchange of data on an established link, for example Internet Protocol (IP) and Internet Control Message Protocol (ICMP).
    3. *Transport layer protocol*, protocols that are implemented on top of the network layer, for example Transmission Control Protocol (TCP), User Datagram Protocol (UDP).
    4. *Application layer protocol*, protocols implemented on top of the transport layer, for example HTTP, FTP, SMPT.
    5. *Logical protocol*, a logical specification of the constraints on the observable behavior of data streams, defined in Chapter 4.
- **Port**
    1. *Physical port*, e.g. an Ethernet port as found on switching devices.
    2. *Transport layer port*, e.g. as used in transport layer protocols TCP and UDP.
    3. *Logical port*, a point of streaming data that can be constrained by a logical protocol.
    4. *Input port*, a logical port from which a protocol may causally receive data.
    5. *Output port*, a logical port to which a protocol may send data.

- **Component**
  1. *Physical component*, such as an electrical or optical wire, antenna, or device.
  2. *Logical component*, a contributor of logical behavior to a communication session in the form of constraints over the session's observable behavior. Owns a set of logical ports.
  3. *Protocol component*, a logical component whose behavior is declared in the form of a logical protocol, acting as a behavioral specification (see Chapter 4).
  4. *Primitive component*, a protocol component which participates in the session's communication by sending and receiving messages through its ports; primitive components are not defined in terms of other components.
  5. *Composite component*, a protocol component constructed as a composition from other protocol components, possibly recursively defined (see Chapter 4).
  6. *Native component*, a logical component of which the logical protocol is unknown. Typically implemented by a process running one some host (see Chapter 3); each connector facilitates a user's application process assuming the role of a native component in a communication. Like primitive components, native components are able to participate in the session's communication by exchanging messages.
- **Node**
  1. *Node*, a vertex in a mathematical graph structure.
  2. *Physical node*, a physical machine in a network.
  3. *Peer*, the entity that is uniquely identified by an Internet address and reachable via IP. A peer is not necessarily one physical machine, as the same physical machine can have multiple IP addresses, and the same IP address can be allocated to multiple physical machines (cf. DNS root server anycast).
  4. *Reo node*, a logical component with variable number of input ports and variable number of output ports, such that at most one input port fires at a time and its data is synchronously replicated to all output ports.
- **Connector**
  1. *Connector handle*, a resource obtained by native applications when executing programs that make use of the application programming interface, defined in Chapter 3.
  2. *Connector*, an abstraction of the communication session between a set of peers. The behavior of a connector is prescribed by a logical protocol.
  3. *Connector component*, a logical component. After a connector is fully connected, we refer to the connector component to mean the composition of all configured logical protocols. The connector component is unique after a connector is fully connected, since all peers are known.
- **Synchronous**
  1. *Synchronous operations* block until some return event has occurred.
  2. *Synchronous events* occur within the same synchronous round of a connector.
  3. Two components must *synchronize* as soon as a port that is channeled from one component's output to the other component's input fires.

- **Imp**

  1. *Interface Message Processor* (historically) a distinct physical node used as a gateway to the ARPANET network.
  2. An entity that implements the interface between a native application and the rest of a connector at run-time. The behavior of a connector is realized through communication among imps.

2.2 Connectors

For applications making use of BSD-style sockets, sockets represent both (a) the application's 'handle' on the communication, and (b) one logical endpoint for two-party message exchange with a networked peer. Applications that require multi-party communication or careful coordination between peers build this functionality into the application itself on top of sockets.

Reowolf introduces the *connector* as an application's 'handle' on communication session involving any number of networked peers, connected by any number of logical, two-party message channels. Most essentially, connectors enable applications synchronously exchanging sets of messages with their peers over the network. However, the real utility of connectors is their extensive configurability, communicated from application to connector as a *protocol description*, expressed in *Protocol Description Language* (PDL), a purpose-built language, specific to the domain of high-level protocol specification. These protocols allow an application to describe their requirements of the session's behavior, effectively (a) delegating the work of multi-party coordination to the connector itself, and (b) explicitly, and unambiguously declaring their requirements in a form that can be shared between connectors, over the network and with middleware. Connectors realize the session layer in the OSI model by realizing the communications of the application-layer ('above') through the utilization of transport-layer resources ('below').

This section gives an overview of the relationship between the application and its connector, and their respective views on, and responsibilities to, realizing the communication session. The connector API which exposes this functionality to user applications is explained in detail in Chapter 3. Reowolf's definitions of 'protocol', and 'session' are given precisely in Chapter 5, while PDL, which expresses protocols which reason about these notions is defined in Chapter 4. The implementation of connectors, facilitating the emergence of the session at runtime is explained in Chapter 6.

Over their lifetime, connectors are in one of two states, corresponding to the sequence of two phases of the session: *setup* and *communication*. To follow, the task of the connector in each of these phases is explained in more detail.

2.2.1 Setup

The connector structure is initialized into the setup phase. For the duration of the phase, the application developer incrementally (a) defines their local view on the session's protocol, and (b) acquire local resources for channel endpoints, along with annotations for how they are realized as transport-layer channels. Both processes occur together by way of a *builder pattern*, allowing the application to refine their session's protocol piece by piece until they are ready to begin communicating. The setup phase ends at the first synchronization barrier: *connection*, the procedure which finalizes the prepared resources, and after which the session's connectors form the *distributed connector runtime*, a system of distributed, cooperating connectors.

During the phase, the application incrementally provides the connector with configuration data. This includes (a) a protocol description data structure, initially parsed from a textual form in PDL, (b) annotations of the relationships between logical ports, forming logical channels, and (c) annotations of the relationships between logical ports and transport-layer endpoints. At the moment the connector connects, the configuration is realized all at once, through its participation in a sequence of distributed meta-control algorithms to establish a consistent distributed state with the session's other connectors.

2.2.2 Communication

A set of connectors enter the communication phase together, synchronously starting the communication session. Henceforth, each connector mediates the communication between its application, and the session at large.

In the communication phase, the distributed connector runtime works to advance the state of the session by identifying and realizing message exchange such that (a) every peer has a consistent view on every interaction, i.e., a message's sender and recipient agree on the message's contents, and (b) interactions don't conflict with the session's protocol.

The role of the application in communication is simply to submit their requirements of the session's behavior, one round at a time. The role of the distributed connector runtime is to make a best effort in identifying and realizing the progression of the session's state one interaction at a time, such that all peers' requirements are satisfied. In this fashion, applications are free to request any communication behavior at all, regardless of the session's configuration in the setup phase. When no satisfactory, consistent interactions can be found, the connector protects the integrity of the session by rejecting the application's most recent requests by raising an error, allowing the application to try again. This relationship between an application and its connector facilitates applications implemented in terms of their local view of the protocol, and trusting the connector to protect the state of the session from all errors that would arise from outside influences.

Chapter 3

Application Programming Interface

This chapter lays out the application programmer interface ('API') for the connector runtime implementation, which provides *connectors* as a generalization of BSD-style sockets for network communication. This interface exposes the functionality of the runtime, whose implementation is first explained in Chapter 6, and expanded with additional features in Chapters 7, and 9. Examples of how this API is used by an application for communications are given in Chapter 10.

Over its development, the connector API has been revised to reflect the development of the implementation. This includes simplification and streamlining of the essential functionality, and extensions to expose features added late in the development cycle of the project. The preliminary API design can be found in Chapter B in the Appendix.

3.1 C Connector API

This section lays out and explains the connector API for the C programming language, reflecting the state of the implementation at the end of the project. A connector is the bridge between a user's application and the session, potentially spanning the network. The design of the connector API is designed to mirror the behavior of *protocol components*, whose abstract definition in relation to sessions is given in Chapter 5, and whose definition is the syntax of PDL is described in Chapter 4.

The header file which exposes the connector API is provided in Section 3.1.1; its contents are explained in Sections 3.1.2 and 3.1.3, approximately partitioning procedures according to state of the connector in which they enable the construction and set-up of the session, and the user's communications, respectively.

Note that the connector API follows the C idiom of encoding error information using signed integer return codes. For all procedures for which this is the case, the return result of `RW_TL_ERR` indicates that a textual description of the error was written to the thread-local buffer, whose contents are readable via a pointer acquired by `reowolf_error_peek`. The precise value of `RW_TL_ERR` is defined in Section 3.1.1 to follow.

3.1.1 Connector Header File

This sections lays out the header file for the C language API in its entirety, for reference in the sections to follow.

```
#ifndef REOWOLF_HEADER_DEFINED
#define REOWOLF_HEADER_DEFINED

#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>

#define RW_BAD_FD -5
#define RW_BAD_SOCKADDR -8
#define RW_CLOSE_FAIL -4
#define RW_CONNECT_FAILED -6
#define RW_LOCK_POISONED -3
#define RW_OK 0
#define RW_TL_ERR -1
#define RW_WOULD_BLOCK -7
#define RW_WRONG_STATE -2

typedef struct Arc_ProtocolDescription Arc_ProtocolDescription;
typedef struct Connector Connector;
typedef uint32_t ConnectorId;
typedef uint32_t U32Suffix;

typedef enum {
  EndpointPolarity_Active,
  EndpointPolarity_Passive,
} EndpointPolarity;

typedef enum {
  Polarity_Putter,
  Polarity_Getter,
} Polarity;

typedef struct {
  ConnectorId connector_id;
  U32Suffix u32_suffix;
} PortId;

typedef struct {
  uint8_t ipv4[4];
  uint16_t port;
} FfiSocketAddr;

const uint8_t *reowolf_error_peek(uintptr_t *len);

// protocol descriptions
Arc_ProtocolDescription *protocol_description_parse(const uint8_t
↪  *pdl, uintptr_t pdl_len);
```

```
Arc_ProtocolDescription *protocol_description_clone(const
 ↪  Arc_ProtocolDescription *pd);
void protocol_description_destroy(Arc_ProtocolDescription *pd);

// connector: setup + miscellaneous
Connector *connector_new(const Arc_ProtocolDescription *pd);
Connector *connector_new_with_id(const Arc_ProtocolDescription *pd,
 ↪  ConnectorId connector_id);
Connector *connector_new_logging(const Arc_ProtocolDescription *pd,
 ↪  const uint8_t *path_ptr, uintptr_t path_len);
Connector *connector_new_logging_with_id(const
 ↪  Arc_ProtocolDescription *pd, const uint8_t *path_ptr, uintptr_t
 ↪  path_len, ConnectorId connector_id);

void connector_add_port_pair(Connector *connector, PortId
 ↪  *out_putter, PortId *out_getter);
int connector_add_net_port(Connector *connector, PortId *port,
 ↪  FfiSocketAddr addr, Polarity port_polarity, EndpointPolarity
 ↪  endpoint_polarity);
int connector_add_udp_mediator_component(Connector *connector, PortId
 ↪  *putter, PortId *getter, FfiSocketAddr local_addr, FfiSocketAddr
 ↪  peer_addr);
int connector_add_component(Connector *connector, const uint8_t
 ↪  *ident_ptr, uintptr_t ident_len, const PortId *ports_ptr,
 ↪  uintptr_t ports_len);
int connector_connect(Connector *connector, int64_t timeout_millis);

void connector_destroy(Connector *connector);
void connector_print_debug(Connector *connector);

// connector: communication
int connector_get(Connector *connector, PortId port);
int connector_put_bytes(Connector *connector, PortId port, const
 ↪  uint8_t *bytes_ptr, uintptr_t bytes_len);
intptr_t connector_next_batch(Connector *connector);
intptr_t connector_sync(Connector *connector, int64_t
 ↪  timeout_millis);
const uint8_t *connector_gotten_bytes(Connector *connector, PortId
 ↪  port, uintptr_t *out_len);

#endif /* REOWOLF_HEADER_DEFINED */
```

3.1.2 Setup

Communications in a persistent session necessitate some steps be taken to prepare the connector for its role in the session before communications can begin. New connectors are initialized given a `Arc_ProtocolDescription` structure, facilitating their configuration. As such, the first procedure a user is likely to call is `protocol_description_parse`, allowing the

initialization of a protocol description structure, given a textual protocol description; for more information about PDL, the textual language for describing protocols, refer to Chapter 4.

Connector structures are initialized using `connector_new`, provided a protocol description; the connector begins in the *setup* state. Note that this procedure is provided alongside variants which allow for some minor meta-configuration of the connector; for example, `connector_new_logging` will create a connector which will dump logging information to a file created with the given path.

As with all components, the *native* component corresponding to the user's application, *owns* a set of ports, i.e., ends of logical communication channels with components in the session. During the setup phase, the user is able to add channels and components to the session, such that they begin communication in sync with the native component itself. Initially, the native has an empty set of owned ports, but has some procedures available for acquiring ownership of newly-created ports (equivalently, we say ports are added to the native's *interface*):

1. `connector_add_port_pair`
   A pair of ports, joined by a newly-created logical channel are created, and their identifiers written via out-pointer to the user's provided variables. If the user provides `NULL` out-pointers, the respective identifier is not written, but the channel is nonetheless created. Both ports are added to the interface of the native. The polarities of both ports are *output*, *input*, in the order they appear in the function declaration. Figure 3.1 shows the result of this procedure for a newly created connector.

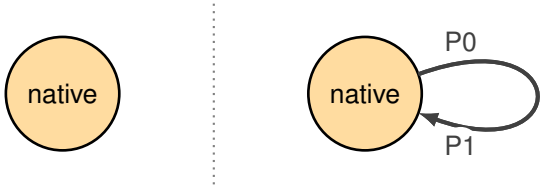


Figure 3.1: Component graph before (left) and after (right) the application uses `connector_add_port_pair` to initialize ports $P0$ and $P1$, linked by a logical channel, with output and input polarities respectively.

2. `connector_add_net_port`
   A single port is created, and written to the user-provided variable via out-pointer, unless the pointer is `NULL`, in which case it is not written to. This procedure facilitates up to two connectors to cooperate in the creation of a logical channel backed by a network channel, by providing both ends separately. To facilitate their rendezvous, the call requires additional transport-layer information, including a socket address (IP address and port integer), as well as the 'role' of the connector in this rendezvous, determining whether the underlying transport channel will be created via a `connect` or `accept` call with the provided socket address. Lastly, the caller can specify which polarity they wish the port to have (input or output). The new port is added to the native component's interface. Note that the transport channel backing the newly-created port is created until the setup procedure ends with `connector_connect`; as such, any malformation of the transport- or logical-channel will not be detected until then. Figure 3.2 shows the result of this procedure for a newly created connector.

3. `connector_add_udp_mediator_component`
   This procedure initializes an intrinsic *udp mediator component* (whose functionality is de-

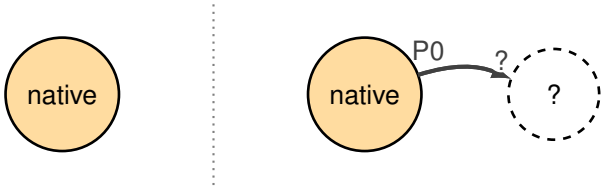


Figure 3.2: Component graph before (left) and after (right) the application uses `connector_add_net_port` to initialize port $P0$, linked with a logical channel to some unknown peer, to be discovered through a transport channel. In this case, the user requests that $P0$ has output polarity.

scribed in Chapter 9), along with two logical channels for sending messages to, and receiving messages from the new component. Per channel, one of the two new ports is added to the interface of the native component, and returned to the user via whichever subset of the user-provided out-pointers' values are not `NULL`. Figure 3.3 shows the result of this procedure for a newly-created connector.

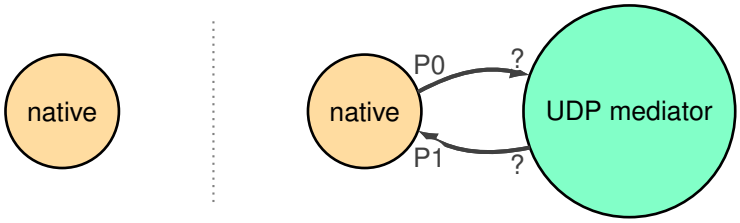


Figure 3.3: Component graph before (left) and after (right) the application uses `connector_add_udp_mediator_component` to initialize an intrinsic UPD mediator component, linked to the native by a pair of newly-created logical channels, returned to the user as the pair of ports $P0$ and $P1$ with output and input polarities respectively.

The real strength of connectors is in the way they facilitate their participants configuring the behavior of the session to follow by instantiating *protocol components*. As far as an application developer is concerned, this can be understood as connectors providing a means for the injection of 'actors' into the session, managed by the connector, and drawing their behavior from the protocol configuration provided in `connector_new` by calling `connector_add_component`. This procedure mirrors the `new` keyword in PDL, which allows protocol components to create other protocol components, and likewise, `connector_add_component` accepts a set of the calling component's ports whose ownership should be moved to that of the newly-created component. Furthermore, the procedure requires the user to provide the identifier of the protocol component, as it occurs in the protocol provided as the connector's configuration (as an ASCII-encoded string). Figure 3.4 shows the result of this procedure in the case of a native component with interface port set $\{P0, P1\}$, creating a component with identifier 'foo', as it appears in the protocol description, and instantiated such that ownership of the native port $\{P1\}$ is moved to the new component.

The setup procedure ends when `connector_connect` completes successfully, indicating that the connector has transitioned to the *communication* phase, and that all port and component resources initialized during the setup phase were completed successfully.

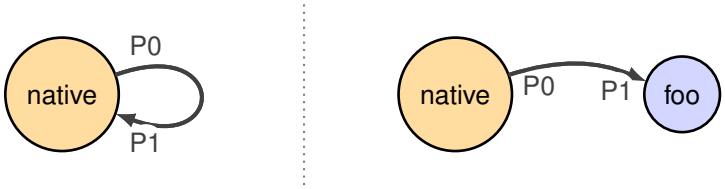


Figure 3.4: Component graph before (above) and after (below) the application uses `connector_add_component` to initialize a component with identifier 'foo', and move ports $\{P1\}$ from the interface of the native to the newly-created component.

Regardless of the connector's phase, it can be destroyed with `connector_destroy` to free the structure's resources, and to exit the session if it was previously connected. If any of a session's participating connectors is destroyed, the other connectors can no longer succeed in any new communications, but they can still access their connectors; as such, they each must be destroyed on their own to avoid leaking memory.

Protocol descriptions passed to connectors are *moved* to the connector's internals, and must not be destroyed. Otherwise, protocol descriptions can be explicitly destroyed using `protocol_description_destroy`. In the event the application would like to instantiate several connectors with the same protocol configuration, `Arc_ProtocolDescription` structures can be replicated safely using `protocol_description_clone`; the returned clone must be destroyed independently of the original. Note that this procedure performs a 'shallow' copy, relying on atomic reference counting to decouple the lifetimes of the replicas; as such, it is advisable to prefer cloning protocol descriptions in this manner to repeatedly parsing the same textual description (which results in a larger memory footprint).

3.1.3 Communication

Once in the communication phase, the connector serves to facilitate the participation of the native component in a sequence of synchronous rounds until the session ends. In each round, the native is able to exchange up to one message (a variable-length byte sequence) through each port in the native's interface (i.e., through each port the native owns). The centerpiece of the communication phase is the procedure `connector_sync`, which is the only procedure involving distributed synchronization, and with each successful completion, pushes the state of the session forward one round. All other procedures can be understood as occurring 'between' rounds, and involve either (a) preparing message exchange operations to be performed in the next round, or (b) reflecting on the result of the previous round.

Calls preceding `connector_sync` work to modify the state of the connector using the *builder pattern*, to incrementally specify the behavior permitted of the ports in the native's interface in the round to follow. This state is encoded as a set of *synchronous batches*, stored by the connector. The initial state of the batches is a single batch (with index zero) containing zero port operations, and after every `connector_sync`, this initial state is reset. The meaning of the batches is the expression of a set of *options*, permitting the connector runtime to realize the outcome as the result of a non-deterministic choice; as such, offering a singleton batch set offers only one possible outcome, and two or more batches ensure that the message exchange operations of at most one batch's operations will succeed, i.e., the success of port operations within different batches is mutually-exclusive, making it safe to alias message data if and only

if the aliasing occurs *across* batches. The following procedures permit the user to modify the connector's batches:

1. `connector_get`
   Given the identifier to an input port *p* in the native component's interface, express the requirement that *p* receives a message in the next round. If successful, *p* is added to the batch with the largest index. The call fails if the native component does not own a port identified with *p*, *p* has output polarity, or the batch with the largest index already contains *p*.

2. `connector_put_bytes`
   Given the identifier to an output port *p* in the native component's interface, express the requirement that *p* sends a given message in the next round. If successful, *p* is added to the batch with the largest index. The call fails if the native component does not own a port identified with *p*, *p* has input polarity, or the batch with the largest index already contains *p*.

3. `connector_next_batch`
   A new, empty batch is added to the set of batches, given an index $i+1$ where i is the current maximal batch index. Effectively, this 'finalizes' the contents of the batch with index i, and makes it such that subsequent calls to `connector_get` and `connector_next_batch` modify batch $i+1$ until the next call to `connector_next_batch` or `connector_sync`.

The `connector_sync` procedure, provided an optional *timeout duration* (in milliseconds), blocks the caller thread until the round succeeds or fails. The timeout duration expresses an urgency to return control to the caller once the timeout has elapsed; however, the consistency of the distributed session is prioritized over this urgency, and so the successful, prompt return of control flow is subject to the behavior of the session's other peers. The return result of the call is an integer, for which non-zero values encode success, and the index of the batch that succeeded. Negative return values express the occurrence of an error, whose precise nature depends on the integer (see the header file for the precise definitions of the error variants). Note that one connector returns an error to some round if and only if all connectors return an error to the same round, though not necessarily the same error. In the event of a recoverable error, the state of the session was successfully *rolled back*, as if the call had not occurred at all, such that it can be retried. The contents of the batches, however, are not restored.

3.2 C Pseudo-Socket API

Section 3.1.2 defined the procedure `connector_add_udp_mediator_component` for use in the setup of sessions in which communications cross the boundary between connectors and sockets. To facilitate the use case of connectors being used *in place of* sockets in a user's application, connectors expose the *pseudo socket API*, provided as a separate header file, `pseudo_socket.h`; Chapter 9 explains these features in detail, along with how they are implemented:

```
#include <sys/socket.h> // defines {sockaddr, socklen_t}
int rw_socket(int domain, int type, int protocol);
int rw_connect(int fd, const struct sockaddr *address, socklen_t
↪  address_len);
int rw_bind(int socket, const struct sockaddr *address, socklen_t
↪  address_len);
```

```
int rw_close(int fd);
ssize_t rw_send(int fd, const void * message, size_t length, int
↪  flags);
ssize_t rw_recv(int fd, void * buffer, size_t length, int flags);
```

Chapter 4

Protocol Description Language

This chapter defines Protocol Description Language (PDL) in a bottom-up fashion, oriented around the structure of its syntax. Section 4.3 and onward, the definition of well-formedness criteria for protocols is introduced. In Chapter 5 to follow, the semantics of protocols is explained in relation to that of connectors, laying the groundwork for the definition of a correct connector implementation.

PDL draws inspiration from general-purpose programming languages C and Java, both pervasively known, and used in the domain of network programming. It is also largely based on Reo, drawing inspiration from its compositional, interaction-based model of concurrency. As such, the reader may wish to refer to the cited specifications or standards for C [ISO18], Java [JSGB00], and textual Reo ('Treo') [DA18].

4.1 Notation

For specifying syntactical structure we employ two notation formats: one for specifying the lexical grammar, another for specifying the abstract syntax tree.

Augmented Backus-Naur Form (ABNF) is a *context-free grammar* specified in RFC5234. It is used for specifying the lexical grammar of the Protocol Description Language. ABNF consists of a number of *production rules*. Each production rule is given a case-insensitive *name* and a defining *expression*.

Expression in ABNF are simple or complex. Simple expressions are formed by a *terminal value* or by a name of a production rule. Complex expressions are formed by the operations: *concatenation* or *alternative* of two expressions, *repetition* with optional minimum and maximum occurrences, *grouping* an expression in parentheses, or an *optional* expression.

```
CRLF            = CR LF
                ; Concatenation of CR and LF rules
HTAB            = %x09
                ; Horizontal tab (encoded value)
LF              = %x0A
                ; Line feed
CR              = %x0D
                ; Carriage return
SP              = " "
```

```
                ; Space character (literal value)
VCHAR           = %x21-7E
                ; Visible characters in 7-bit ASCII
WSP             = SP / HTAB
                ; Alternative of SP and HTAB rules
ALPHA           = %x41-5A / %x61-7A
                ; Value range for letters A-Z or a-z
DIGIT           = %x30-39
                ; Value range for digits 0-9
```

The terminal values are given either directly (e.g. `%x09`) or by specifying a *value range* (e.g. `%x21-7E`). The above example shows rules which are commonly used in ABNF. The sentence following a semicolon are comments that describe what the expression above it matches as terminal value.

We specify the abstract syntax tree using pseudo-code that should feel familiar to readers who have seen Java, C++ or the Interface Description Language (IDL) published by Object Management Group (OMG) before. We focus on specification in an object-oriented manner: elements in the abstract syntax tree are objects. Using this pseudo-code, we define *interfaces* which may inherit from (multiple) other interfaces, and *(abstract) classes* that may extend one super class and inherit from (multiple) interfaces, and have *methods* that provide access to its *attributes*.

```
interface InputPosition {
    String getFilename();
    int getLine();
    int getColumn();
    int getOffset();
}
class Element {}
class SyntaxElement extends Element {
    InputPosition getPosition();
}
```

Above is an example interface and two classes. These represent the position of a source file, and a syntactical element that has a position in a source file. The first interface, `InputPosition`, has four attributes: filename, line number, column number, and absolute offset. The second interface, `SyntaxElement`, has one attribute: a reference to an object that implements the first interface. We shall assume that we treat returned objects as *immutable*, not to be modified by users of these interfaces and classes.

In some superficial way, our interface definitions can also be seen as a context-free grammar. Each interface or class is a production rule. Inheritance specifies alternative choices. Interface attributes are the names of the concatenation of other productions.

4.2 Context-Free Grammars

4.2.1 Lexical Analysis

The lexical analysis of the Protocol Description Language consists of two conceptual layers: lower-level character input handling and higher-level syntax grammar. Processing of these two

layers can happen in separate phases, or at the same time. We specify the lexical analysis by defining a grammar. Our goals in specifying this grammar are: unambiguous parsing and keeping implement to a minimum.

Input Handling

The following rules are used mainly to process character input data, filtering out comments and white space. We assume the input data is consumed per octet (8 bit). Input data matching these rules is ignored and not significant.

```
; The following terminals are ignored by the parser
cwb             = cw
                ; cwb must match a word boundary
cw              = comment / WSP / newline
comment         = line-comment / block-comment
newline         = CRLF / LF
                ; Special treatment of new lines
line-comment    = "//" *(WSP / VCHAR) newline
block-comment   = "/*" *block-no-eob "*/"
                ; A block comment may contain anything except "*/"
block-no-eob    = "*" (%x00-2E / %x30-FF)
                / (%x00-29 / %x2B-FF)
                ; Content of block may include non-visible characters
```

The difference between rule `cwb` and `cw` is that the former rule is a marker that serves as a reminder that we expect a word boundary; both rules consume the same terminals. Note that `cw` stands for *comment* or *whitespace*, but a newline is also accepted.

We treat new lines specially. We accept either a line feed, or a carriage return immediately followed by a line feed. Input handling also accepts just carriage returns as new line, but must greedily consume a line feed if it is immediately following a carriage return. With respect to the position in the input data, all three forms result in the line count to increase by one, and the column to be reset to zero.

A comment is either a line comment that is followed only by visible characters in the ASCII range, or a block comment that may contain arbitrary binary data except for the `"*/"` sequence that marks the end of the comment. A line comment ends after the end of line is encountered.

```
; Common symbols
binry-operator = "||" / "&&" / "|" / "^" / "&" / "==" /
                 "!=" / "<=" / ">=" / "<" / ">" / "<<" /
                 ">>" / "+" / "-" / "*" / "/" / "%"
assgn-operator = "=" / "*=" / "/=" / "%=" / "+=" / "-=" /
                 "<<=" / ">>=" / "&=" / "^=" / "|="
unary-operator = "++" / "--" / "+" / "-" / "~" / "!"

; Common tokens
char-constant  = "'" 1*(SP / %x21-26 / %x28-7E) "'"
               ; Character literals are arbitrary length
int-constant   = DIGIT *(DIGIT/"a"/"b"/"c"/"d"/"e"/"f"/"x")
               ; Integer literals are arbitrary length
ident          = 1*ALPHA *(ALPHA / DIGIT / "_")
```

```
                ; An identifier is an alphanumeric sequence
                ; that starts with an alphabetical character.
```

We commonly recognize certain characters as *binary operators*, *assignment operators* or *unary operators*. Later syntactic structures make sure to specify operator precedence and associativity. Remark that `"+"` and `"-"` can be confused, as both are binary and unary operators.

We also recognize constants and identifiers. *Character constants* may appear as at least one character, enclosed within single quotes and restricted to the visible characters in the ASCII range except for the single quote itself. Similarly, *integer constants* are recognized whenever a digit is encountered and hexadecimal constants such as `0xEA` and octal constants such as `076` can be recognized: but processing of the constant value itself happens later. Identifiers, representing names within a protocol description, must start with an alphabetic character, but may be followed by any alphanumeric sequence of characters including underscores.

```
keywords        = "import" / "composite" / "primitive" / "in" / "out" /
                  "channel" / "msg" / "boolean" / "byte" / "short" /
                  "int" / "long" / "null" / "true"/ "false" /
                  "if" / "else" / "while" / "break" / "continue" /
                  "synchronous" / "return" / "assert" /
                  "goto" / "skip" / "new"
builtin         = "put" / "get" / "fires" / "create"
```

We recognize *keywords* and *built-in* functions. Keywords are used to structure definitions, parameters, types and statements. The important keywords are `"composite"`, `"primitive"` and `"synchronous"`. Built-in functions are special. When displaying protocol descriptions, keywords and built-ins are typically shown in a boldface type.

```
type            = "in" / "out" / "msg" / "boolean" /
                  "short" / "int" / "long" / ident
                ; A type is a keyword or an identifier
method          = builtin / ident
                ; A method is a built-in or symbolic
field           = "length" / ident
                ; A field is length or an identifier
```

We further distinguish three classes of identifiers, each permitting different keywords. A type is either an *input*, *output*, *message*, *boolean*, integer of various sizes, or otherwise identified. A *method* is either a built-in function or symbolic: the meaning of the latter depends on its context, and can refer either to a symbolic function or a component. A field is an identifier, but we recognize `"length"` as a special field. `"length"` is not a keyword, however.

Besides the specified keywords and built-ins, identifiers should not be equal to other keywords or built-ins to avoid confusion. The grammar does not explicitly formulates this requirement, but a parser must avoid keywords or built-ins to be used as identifiers.

Syntax Structure

Parsing a protocol description begins with the *file* production rule.

```
file            = *cw *(pragma *cw) *(import *cw) 1*(symbol-def *cw)
                ; A file comprises one or more definitions
```

```
pragma          = "#" [VCHAR *(WSP / VCHAR)] newline
                ; Pragma until end of line
import          = "import" 1*cw (ident *("." ident)) *cw ";"
                ; Import declarations
```

Each file consists of a number of *pragmas* that begin with `"#"`. These are used to convey version information of the Protocol Description Language, to allow for future language changes. Following the pragmas, zero or more *imports* of qualified identifiers, being identifiers separated by dots. A reverse domain name convention for imports is used, similar to Java packages. An application may register PDL files using these qualified identifiers, as a basic module system for protocol descriptions. Imports that are starting with the `std` identifier are reserved for predefined standard imports, and cannot be registered.

```
symbol-def      = fun-def / component-def
                ; Symbol definition
fun-def         = type-annot 1*cw ident *cw fparams *cw block
component-def   = composite-def / primitive-def
                ; Component definition is either composite or primitive
composite-def   = "composite" 1*cw ident *cw fparams *cw block
primitive-def   = "primitive" 1*cw ident *cw fparams *cw block
```

The rest of a protocol description consists of *symbol definitions*. Every symbol definition consists of an identifier (the *symbol*), *formal parameters*, and a *block*. There are two kinds of symbol definitions: *function definitions* and *component definitions*. A function definition additionally consists of a return type. A component definition is either composite or primitive. Identifiers of symbol definitions must not be equal to built-ins.

```
type-annot      = type [*cw "[]"]
var-decl        = type-annot 1*cw ident
                ; Variable declaration (optionally an array)
fparams         = "(" *cw [var-decl *(*cw "," *cw var-decl)] *cw ")"
                ; Formal parameter list
```

The formal parameters of a symbol definition are given as a list within parentheses. Each formal parameter is of some type. Each formal parameter declares a variable that is in the scope of the block of a symbol definition. Types may be arrays, as indicated by `"[]"` following the type, but arrays of arrays are not allowed.

```
block           = "{" *(*cw (channel-decl/mem-decl)) *(*cw stmt) *cw "}"
channel-decl    = "channel" 1*cw ident *cw "->" *cw ident *cw ";"
mem-decl        = type-annot 1*cw ident *cw "=" *cw expr
                  *(*cw "," *cw ident *cw "=" *cw expr) *cw ";"
```

A block consists of zero or more *local declarations*, followed by zero or more statements. A local declaration is either a *channel declaration* or a *memory declaration*. Channel declarations start by the `"channel"` keyword, followed by two identifiers separated by an arrow `"->"`. Memory declarations start by a type annotation, followed by one or more identifiers with an *expression* that designate the initial value of the memory. The local variable declarations declare variables that are in scope of the statements and expressions in the same block.

```
stmt            = block
                / ident *cw ":" *cw stmt
                / "if" *cw pexpr *cw stmt [*cw "else" *cwb stmt]
                / "while" *cw pexpr *cw stmt
                / "break" *cwb [ident *cw] ";"
                / "continue" *cwb [ident *cw] ";"
                / "synchronous" *cw (fparams *cw stmt / block)
                / "assert" *cwb expr *cw ";"
                / "return" *cwb expr *cw ";"
                / "goto" 1*cw ident *cw ";"
                / "skip" *cw ";"
                / "new" 1*cw method-expr *cw ";"
                / expr *cw ";"
```

A *statement* is either a block, a *skip* statement, a *labeled* statement, an *if* statement, a *while* statement, a *break* statement, a *continue* statement, a *synchronous* statement, a *return* statement, an *assert* statement, a *goto* statement, a *new* statement or an expression. Except for blocks, labels and expressions, the first keyword designates the kind of statement. The `"else"`, `"break"`, `"continue"`, `"synchronous"`, `"return"` and `"assert"` keywords must be at a word boundary, to prevent matching a following keyword or identifier without white space in between. To prevent confusion between variable declarations and expressions, synchronous statements without formal parameters are always followed by a block statement.

A statement may be labeled so break, continue and goto statements can refer to the label.[1] These statements are collectively refered to as control flow disruption statements. Not all control flow disruptions are valid. Labeled break and continue statements are also found in Java. The labeled goto statement is found in C.

There is an explicit skip statement, in contrast to languages such as C, C++ and Java, because in PDL empty blocks are not allowed. The *else-branch* of an if statement is optional, but is eagerly associated to avoid ambiguity.

```
if(true) if(true) put(x); else put(y);
if(true){if(true) put(x); else put(y);}
if(true){if(true) put(x);}else put(y);  // !!!

if(i<j) if(j<k) if(k<n) put(x); else put(y); else put(z);
if(i<j){if(j<k){if(k<n) put(x); else put(y);}else put(z);}
if(i<j){if(j<k){if(k<n) put(x);}else put(y);}else put(z); // !!!
```

Above example consists of two triples of similar statements. The first two statements are parsed in the same way, but this differs from the third statement. Eager association of the else-branch means that inner if-statements associate to the else branch; accolades can be used to change the standard grouping, as is done in the third statement.

```
pexpr           = "(" *cw expr *cw ")"
expr            = assgn-expr
assgn-expr      = cond-expr [*cw assgn-operator *cw expr]
cond-expr       = concat-expr [*cw "?" *cw expr *cw ":" *cw expr]
concat-expr     = lor-expr *(*cw "@" *cw lor-expr)
lor-expr        = land-expr *(*cw "||" *cw land-expr)
```

[1] See the Structured Programming with `goto` Statements article by Donald E. Knuth (1974) for technical discussion.

```
land-expr      = bor-expr *(*cw "&&" *cw bor-expr)
bor-expr       = xor-expr *(*cw "|" *cw xor-expr)
xor-expr       = band-expr *(*cw "^" *cw band-expr)
band-expr      = eq-expr *(*cw "&" *cw eq-expr)
eq-expr        = rel-expr *(*cw ("=="/"!=") *cw rel-expr)
rel-expr       = shift-expr *(*cw ("<="/">="/"<"/">") *cw shft-expr)
shift-expr     = add-expr *(*cw ("<<"/">>") *cw add-expr)
add-expr       = mul-expr *(*cw ("+"/"-") *cw mul-expr)
mul-expr       = unary-expr *(*cw ("*"/"/"/"%") *cw unary-expr)
prefix-expr    = *(unary-operator *cw) postfix-expr
postfix-expr   = primary-expr *(*cw postfix)
postfix        = "++" / "--" / index / select
index          = "[" *cw expr [*cw ":" *cw expr] *cw "]"
select         = "." *cw field
primary-expr   = pexpr/constant-expr/method-expr/array-expr/ident
constant-expr  = int-constant/char-constant/"null"/"true"/"false"
method-expr    = method *cw "(" *cw [expr *(*cw "," *cw expr)] *cw ")"
array-expr     = "{" *cw [expr *(*cw "," *cw expr)] *cw "}"
```

An *expression* is a C-like expression. Some expressions are surrounded by parentheses, but this is to avoid ambiguity or to override operator precedence. An *assignment expression* allows assignment of right-hand side expressions to left-hand side variables. A *conditional expression* tests some expression, and has a true expression after `"?"` and a false expression after `":"`. We consider, generally speaking, *binary expressions* and *unary expressions*. The binary expressions consists of a binary operator and left and right operand expressions, the unary expressions of a unary operator and an operand expression. *Index expressions* and *slice expressions* represent array access and *select expressions* represent field access.

See also Table 4.1 for operator precedence and associativity.[2] The arity of an operators determines the number of operands. Operators with a higher precedence are parsed as the operands of operators with a lower precedence. For example, `1+2*3` is parsed as `1+(2*3)`, where the expression `2*3` is an operand of the addition operator, since multiplication has a higher precedence than addition.

Conditional operators are parsed specially. The true expression of a conditional expression is parsed as if surrounded by parentheses. Otherwise, it is right-to-left associative. For example, `x?y?z:w:w` is parsed as `x?(y?z:w):w`, and `x?y:z?w:w` is parsed as `x?y:(z?w:w)` and *not* as `(x?y:z)?w:w`.

Constant expressions, *method expressions*, *array construction expressions* and *variable expressions* are primary expressions. Method expressions are parsed specially. A method consists of an identifier *m* followed by parentheses in which zero or more comma-separated argument expressions occur. Array construction expressions also consists of zero or more comma-separated element expressions, but surrounded by accolades instead of parentheses. Array constructions are used to give variable number of method arguments.

Harmless ambiguities

The statement `if (x) {}` is ambiguous, as it can be parsed either as an if statement with variable `x` as test expression and an empty block statement as true body, or with an expression statement as true body that is an array construction expression without any elements. A parser

[2] Compare with `https://en.wikipedia.org/wiki/Operators_in_C_and_C++#Operator_precedence`

Operator	Name	Fixity	Arity	Level	Associativity
(_)	parentheses	circumfix	unary	1	n/a
_++	increment	postfix	unary	2	n/a
_--	decrement	postfix	unary	2	n/a
_[_]	array indexing	suffix	binary	2	left-to-right
_._	field access	infix	binary	2	left-to-right
++_	increment	prefix	unary	3	n/a
--_	decrement	prefix	unary	3	n/a
+_	positive	prefix	unary	3	n/a
-_	negative	prefix	unary	3	n/a
!_	logical not	prefix	unary	3	n/a
~_	bitwise not	prefix	unary	3	n/a
_*_	multiplication	infix	binary	4	left-to-right
_/_	division	infix	binary	4	left-to-right
_%_	remainder	infix	binary	4	left-to-right
_+_	addition	infix	binary	5	left-to-right
_-_	subtraction	infix	binary	5	left-to-right
_<<_	bitwise shift left	infix	binary	6	left-to-right
_>>_	bitwise shift right	infix	binary	6	left-to-right
_<=_	less or equal	infix	binary	7	left-to-right
_<_	less than	infix	binary	7	left-to-right
_>=_	greater or equal	infix	binary	7	left-to-right
_>_	greater than	infix	binary	7	left-to-right
_==_	equal	infix	binary	8	left-to-right
_!=_	not equal	infix	binary	8	left-to-right
_&_	bitwise and	infix	binary	9	left-to-right
_^_	bitwise xor	infix	binary	10	left-to-right
_|_	bitwise or	infix	binary	11	left-to-right
_&&_	logical and	infix	binary	12	left-to-right
_||_	logical or	infix	binary	13	left-to-right
_?_:_	conditional	mixfix	ternary	14	special
_=_	assignment	infix	binary	14	right-to-left
_*=_	multiplied "	infix	binary	14	right-to-left
_/=_	divided "	infix	binary	14	right-to-left
_%=_	remained "	infix	binary	14	right-to-left
_+=_	added "	infix	binary	14	right-to-left
_-=_	subtracted "	infix	binary	14	right-to-left
_<<=_	shifted left "	infix	binary	14	right-to-left
_>>=_	shifted right "	infix	binary	14	right-to-left
_&=_	bitwise and'd "	infix	binary	14	right-to-left
_^=_	bitwise xor'd "	infix	binary	14	right-to-left
_|=_	bitwise or'd "	infix	binary	14	right-to-left

Table 4.1: Operator arity, precedence, and associativity. Shown in ascending level. A lower level means a higher precedence and vice versa. The underscores are operands and can be complex expressions or simple expressions: a constant or variable.

prefers the first alternative, but the second alternative is harmless as the array construction expression is side-effect free and has the same behavior as an empty block statement.

Other ambiguous language constructs may be present but currently unknown.

4.2.2 Abstract Syntax Tree

During the parsing of input data, an abstract syntax tree is constructed. The resulting tree does not correspond 1-to-1 to the production rules of the grammar given in Section 4.2.1, and some information originally present is removed (such as parentheses and implicit precedence).

```
class Element {}
class SyntaxElement extends Element {
    InputPosition position;
}
class ExternalElement extends Element {
    Import external;
}
```

We shall describe the hierarchy of elements of the abstract syntax tree. Certain elements in the syntax tree are imported from an external source. Each element is either present from the original source at some position (see Section 4.1) or external.

```
interface Identifier {
    String getValue();
}
class ExternalIdentifier extends ExternalElement
implements Identifier {
    String value;
}
class SourceIdentifier extends SyntaxElement
implements Identifier {
    String value;
}
```

Identifiers are either externally provided or present in the source file. An externally provided identifier results from importing symbol declarations. If an identifier is present in the source file, it has a position through inheritance from syntactical element. This pattern appears for other elements too.

```
interface Type {
    byte TYPE_INPUT = 1;
    byte TYPE_OUTPUT = 2;
    byte TYPE_MESSAGE = 3;
    byte TYPE_BOOLEAN = 4;
    byte TYPE_BYTE = 5;
    byte TYPE_SHORT = 6;
    byte TYPE_INT = 7;
    byte TYPE_LONG = 8;
    byte TYPE_SYMBOLIC = -1;
    byte getSort();
```

```
}
interface SymbolicType extends Type {
    Identifier getIdentifier();
}
class BasicType implements Type {
    byte sort;
}
class SourceType extends SyntaxElement implements Type {
    byte sort;
}
class ExternalSymbolicType extends ExternalElement
        implements SymbolicType {
    ExternalIdentifier identifier;
}
class SourceSymbolicType extends SourceType
        implements SymbolicType {
    SourceIdentifier identifier;
}
```

There are ten types, the first nine are types with a corresponding keyword. The tenth type is symbolic and has the additional identifier attribute.

```
interface TypeAnnotation {
    Type getType();
    boolean isArray();
}
class BasicTypeAnnotation implements TypeAnnotation {
    Type type;
    boolean array;
}
class SourceTypeAnnotation extends SyntaxElement
        implements TypeAnnotation {
    SourceType type;
    boolean array;
}
```

In variable declarations, such as local variable declarations and formal parameter declarations, a type annotation is provided that specifies the type and whether the variable is an array.

```
abstract class Constant extends SyntaxElement {}
class NullConstant extends Constant {}
class TrueConstant extends Constant {}
class FalseConstant extends Constant {}
class CharacterConstant extends Constant {
    String value;
}
class IntegerConstant extends Constant {
    String value;
}
```

Constant expressions, as present in the source, are either a Boolean constant, a character constant, or an integer constant. The value attribute of the latter is the raw character string from the input file: its integer value has to be calculated later. Not all character strings are valid constants: for example, `093` would be accepted by the parser but is an invalid octal number.

```
abstract class Method extends SyntaxElement {}
class BuiltinMethod extends Method {
    final static byte METHOD_PUT = 0;
    final static byte METHOD_GET = 1;
    final static byte METHOD_FIRES = 2;
    final static byte METHOD_CREATE = 3;
    byte sort;
}
class SymbolicMethod extends Method {
    SourceIdentifier identifier;
    SymbolDeclaration declaration;
}
```

There are two kinds of methods as they appear in the source: built-ins and symbolic methods. The terminology of *method* refers to either a function or a component, depending on its context in which it is used. The declaration a symbolic method refers to is resolved after parsing and applying grammar rules.

```
class ProtocolDescription extends SyntaxElement {
    List<Pragma> pragmas;
    List<Import> imports;
    List<SymbolDefinition> symbolDefinitions;
    List<SymbolDeclaration> symbolDeclarations;
}
class Pragma extends SyntaxElement {
    String value;
}
class Import extends SyntaxElement {
    String value;
}
```

The root element of a protocol description consists of three attributes: a list of pragmatics, as they appear at the beginning of the file, a list of qualified imports, and a list of symbol definitions. The import value attribute is the qualified identifier. After resolving the imports, all symbol declarations are known, relative to which symbolic methods are resolved.

```
interface SymbolDeclaration {
    Identifier getIdentifier();
    List<TypeAnnotation> getParameterTypes();
}
interface ComponentDeclaration extends SymbolDeclaration {}
interface FunctionDeclaration extends SymbolDeclaration {
    TypeAnnotation getReturnTypeAnnotation();
}
abstract class ExternalSymbolDeclaration extends ExternalElement
```

```
implements SymbolDeclaration {
    ExternalIdentifier identifier;
    List<TypeAnnotation> parameterTypes;
}
class ExternalComponentDeclaration extends ExternalSymbolDeclaration
implements ComponentDeclaration {}
class ExternalFunctionDeclaration extends ExternalSymbolDeclaration
implements FunctionDeclaration {
    TypeAnnotation returnTypeAnnotation;
}
abstract class SymbolDefinition extends SyntaxElement
implements SymbolDeclaration, VariableScope {
    SourceIdentifier identifier;
    List<FormalParameter> formalParameters;
    BlockStatement block;
}
class FunctionDefinition extends SymbolDefinition
implements FunctionDeclaration {
    SourceTypeAnnotation returnTypeAnnotation;
}
abstract class ComponentDefinition extends SymbolDefinition
implements ComponentDeclaration {}
class CompositeDefinition extends ComponentDefinition {}
class PrimitiveDefinition extends ComponentDefinition {}
```

A symbol declaration consists of an identifier and a signature, being a list of type annotations of the parameters. A symbol declaration is either a component declaration or a function declaration. Function declarations moreover have a return type annotation. Symbol declarations are either imported from some external source, or are given in the source by a symbol definition. Symbol definitions consists of formal parameters, which includes the names of parameters. The signature of a symbol definition is obtained by stripping the formal parameter names, leaving only their types.

```
interface VariableScope {
    VariableScope getParent();
    List<? extends VariableDeclaration> getVariableDeclarations();
}
abstract class VariableDeclaration extends SyntaxElement {
}
class FormalParameter extends VariableDeclaration {
    SourceTypeAnnotation typeAnnotation;
    SourceIdentifier identifier;
}
class LocalVariableDeclaration extends VariableDeclaration {}
class ChannelDeclaration extends LocalVariableDeclaration {
    SourceIdentifier from_identifier;
    SourceIdentifier to_identifier;
}
class MemoryDeclaration extends LocalVariableDeclaration {
    SourceTypeAnnotation typeAnnotation;
```

```
    SourceIdentifier identifier;
    Expression initial;
}
```

A variable declaration consists of a type annotation and a name. Formal parameters and local variable declarations are variable declarations. There are two local variable declarations: channel declarations, and memory declarations. The former consists of two identifiers, representing each channel end. The latter consists of an initialization expression that gives the memory its initial value. Moreover, we have the notion of variable scope: a symbol definition and a block statement introduce a new variable scope. A variable scope has an optional parent variable scope, and consists of a list of the variable declarations declared in that scope.

```
interface LabelScope {
    BlockStatement getParentBlock();
}
abstract class Statement extends SyntaxElement {}
class BlockStatement extends Statement
implements VariableScope, LabelScope {
    List<LocalVariableDeclaration> locals;
    List<Statement> statements;
    VariableScope parent;
    List<LabeledStatement> labels;
}
class SkipStatement extends Statement {}
class LabeledStatement extends Statement {
    SourceIdentifier label;
    Statement body;
}
class IfStatement extends Statement {
    Expression test;
    Statement trueBody;
    Statement falseBody;
}
class WhileStatement extends Statement {
    Expression test;
    Statement body;
}
class BreakStatement extends Statement {
    SourceIdentifier label;
    WhileStatement target;
}
class ContinueStatement extends Statement {
    SourceIdentifier label;
    WhileStatement target;
}
class SynchronousStatement extends Statement
implements VariableScope {
    List<FormalParameter> formalParameters;
    Statement body;
    BlockStatement parent;
```

```
}
class ReturnStatement extends Statement {
    Expression expression;
}
class AssertStatement extends Statement {
    Expression expression;
}
class GotoStatement extends Statement {
    SourceIdentifier label;
    LabeledStatement target;
}
class NewStatement extends Statement {
    MethodCallExpression expression;
}
class ExpressionStatement extends Statement {
    Expression expression;
}
```

A statement is either a block statement, a skip statement, a labeled statement, an if statement, a while statement, a break statement, a continue statement, a synchronous statement, a return statement, an assert statement, a goto statement, a new statement, or an expression statement.

A block statement consists of a list of local variable declarations and a list of statements. The parent variable scope in which the block statement is situated is either a symbol declaration, a synchronous statement or another block statement. The parent block of a block statement is nothing if the parent variable scope is a symbol declaration, just the block statement if that is its parent variable scope, or the parent variable scope of the synchronous statement if that is the blocks parent variable scope. Moreover, a block consists of a list of labeled statements that are directly nested under it. A block statement can have zero or more directly nested statements.

A skip statement can be used in place of an empty block statement.

An if statement consists of a test expression and a true body statement or a false body statement.

A while statement consists of a test expression and a body statement.

Break and continue statements have an optional label; by label resolution the while statement that they refer to is resolved.

A synchronous statement is a variable scope, as it may introduce zero or more formal choice parameters. A synchronous statement always has a block statement as its parent variable scope (see Section 4.3.1), not to be confused with the fact that synchronous statements are not necessarily directly nested under a block statement.

A return statement consists of an expression that describes the return value; and, an assert statement consists of an expression checked for truth.

A goto statement consists of a label, and the labeled statement it refers to is resolved by label resolution. In contrast to break and continue statements, goto statements may point to labeled statements that are in label scope.

A new statement consists of a method call expression; its corresponding symbol declaration is resolved by method resolution.

An expression statement consists of an expression.

```
abstract class Expression extends SyntaxElement {
}
```

```
enum AssignmentOperator {
    SET, MULTIPLIED, DIVIDED, REMAINED, ADDED, SUBTRACTED,
    SHIFTED_LEFT, SHIFTED_RIGHT, BITWISE_ANDED, BITWISE_XORED,
    BITWISE_ORED
}
class AssignmentExpression extends Expression {
    Expression leftExpression;
    AssignmentOperator operator;
    Expression rightExpression;
}
class ConditionalExpression extends Expression {
    Expression test;
    Expression trueExpression;
    Expression falseExpression;
}
enum BinaryOperation {
    CONCATENATE, LOGICAL_OR, LOGICAL_AND, BITWISE_OR, BITWISE_XOR,
    BITWISE_AND, EQUALITY, INEQUALITY, LESS_THAN, GREATER_THAN,
    LESS_THAN_EQUAL, GREATER_THAN_EQUAL, SHIFT_LEFT, SHIFT_RIGHT,
    ADD, SUBTRACT, MULTIPLY, DIVIDE, REMAINDER
}
class BinaryExpression extends Expression {
    BinaryOperation operation;
    Expression leftExpression;
    Expression rightExpression;
}
enum UnaryOperation {
    POSITIVE, NEGATIVE, BITWISE_NOT, LOGICAL_NOT, PRE_INCREMENT,
    PRE_DECREMENT, POST_INCREMENT, POST_DECREMENT
}
class UnaryExpression extends Expression {
    UnaryOperation operation;
    Expression expression;
}
class IndexingExpression extends Expression {
    Expression subject;
    Expression index;
}
class SlicingExpression extends Expression {
    Expression subject;
    Expression fromIndex;
    Expression toIndex;
}
class SelectExpression extends Expression {
    Expression subject;
    SourceIdentifier field;
}
class ConstantExpression extends Expression {
    Constant value;
```

```
}
class ArrayConstructExpression extends Expression {
    List<Expression> elements;
}
class MethodCallExpression extends Expression {
    Method method;
    List<Expression> arguments;
}
class VariableExpression extends Expression {
    SourceIdentifier identifier;
    VariableDeclaration declaration;
}
```

An expression is either an assignment expression, a conditional expression, a binary expression, a unary expression, an array indexing expression, an array slicing expression, a select expression, a constant, an array construction expression, a method call expression or a variable.

An assignment expression assigns the left-hand side expression the value described by the right-hand side expression. The assignment operator employed may be either setting or modifying. A set assignment operator indicates that the value of the right-hand side is stored to what the left-hand side expression describes, discarding the old value of that left-hand side expression. A modifying assignment operator moreover takes into account the old value of the left-hand side expression. A multiplied assignment takes the old value of the left-hand side and multiplies it by the value of the right-hand side; the result is stored to what the left-hand side expression describes. In a similar way, we have divided (stores the result of dividing the old value by the right value), remained (stores the remainder of dividing the old by the right value), added, subtracted, shifted left, shifted right, performing a bitwise AND operation, bitwise XOR operation, or bitwise OR operation.

A conditional expression consists of a test expression, and a true and false expression. The truth of the value described by the test expression determines whether the conditional expression evaluates the true or the false expression.

A binary expression consists of a binary operator and takes a left and a right expression. The value that a binary expression describes is determined by the value of its directly nested expressions. The binary operations are: array concatenation, logical and short-circuit OR, logical and short-circuit AND, bitwise OR, bitwise XOR, bitwise AND, equality or inequality, arithmetic comparison operations, bitwise shift operations, and the usual arithmetic operations.

A unary expression consists of a unary operator and an expression. The value of a unary expression is determined by the value of its directly nested expression. The unary operators are: numerical positive, numerical negative, bit flipping, logical negation, increment and decrement either giving back the old or the new value.

An array indexing expression takes a subject array and an index. It describes looking up the value of the subject array at the offset described by the index. An array slicing expression takes a subject array, and gives as result another array limited between the from index and the to index. The from index is inclusive, the to index is exclusive, meaning that the length of the resulting slice is $to - from$.

A select expression selects a field of its subject. Fields are identifiers. A good example is the lenght field of an array, indicating the number of elements of the array.

A constant expression describes the value that its constant represent.

An array construction expression takes a list of directly nested expressions and constructs an array out of it. A method call expression takes a method and a list of directly nested expressions

as arguments. The method is either a built-in or a symbolic method. In the latter case, the corresponding symbol declaration is resolved by method resolution.

A variable expression consists of a variable identifier. As the result of variable resolution, the variable declaration corresponding to the variable is linked to the variable expression.

4.3 Grammar Rules

After lexical analysis an abstract syntax tree is constructed. However, the parsing process is not yet completed: we check a number of rules to determine the *well-formedness* of the protocol description. Moreover, we perform *symbol resolution* by processing the imported symbol declarations and resolving symbolic references. In a similar manner, but local to each definition, we also perform *variable resolution* for variable declarations. Finally, we perform *type checking*.

One can make use of a general recursor for traversing the abstract syntax tree. The recursor is specialized to implement each rule. As the recursor traverses the tree it maintains a state, based on which a decision can be made to explore the tree further, or to raise a syntax error.

4.3.1 Well-formedness

The first list of rules that we check is well-formedness of the protocol description. These rules are checked in the order as given here, so later rules can assume that the protocol description is well-formed according to the previous rules. Some checks are semantically motivated: the protocol description may have no clear meaning if these checks are not performed.

Nested synchronous statements

In the block of every composite definition or function definition, no synchronous statements may occur either directly or nested under other statements. Within a primitive definition, no synchronous statement occurs either directly or nested under any other synchronous statement. The example below demonstrates three violations of this rule:

```
composite main(in a, out b) {
    new other(a,b);
    synchronous skip; // illegal
}
int fun(int x) {
    synchronous skip; // illegal
    return x;
}
primitive other(in a, out b) {
    while (fun(1) > 1) {
        synchronous { // legal
            synchronous skip; // illegal
}   }   }
```

Invalid variable declarations

Node declarations must not occur within function definitions or primitive definitions. A formal parameter of a synchronous statement of input or output type is invalid. The example below demonstrates a violation:

```
primitive dir(in a, out b) {
    while (true) {
        channel x -> y; // illegal declaration
        synchronous skip;
}   }
```

Function return statement

Primitive and composite definitions do not have a return statement. The block statement of a function definition must return. A statement returns if it is a return statement, or if it is a goto statement, or if it is a block statement where its last statement returns, or if it is an if statement where both branches returns, or if it is a while or labeled statement where its statement returns. All other statements do not return. The following example demonstrates a complex function that does not return:

```
int myfun(int x) {
    if (x > 0) {
        while (x > 0) {
            x--;
            if (x == 0) skip; // illegal
            else return x;
        }
    } else {
        int y = 0;
        label:
        if (y >= 0) {
            goto label;
        } else {
            y = 5;
            return myfun(x + 1);
}   }   }
```

Valid occurrences of built-ins

Except for `create` are built-ins not allowed outside of synchronous blocks. This implies that these built-ins are not allowed inside composite or function definitions, since they do not have synchronous blocks.

```
primitive main(in a, out b) {
    int x = 0;
    msg y = create(0); // legal
    while (x < 10) {
        y = get(a); // illegal
        synchronous {
```

```
            y = get(a); // legal
}   }   }
```

Invalid assignment expressions

The left-hand side of an assignment expression must be an assignable expression. A variable expression is assignable. An indexing expression is assignable. A slicing expression is assignable. A field expression is assignable. All other expressions are not assignable.

Invalid indexing and slicing expressions

The subject of an indexing expression must be an indexable expression. The subject of a slicing expression must also be an indexable expression. A variable expression is indexable. A concatenation expression is indexable. An array construction expression is indexable. A slicing expression is indexable. A select expression is indexable. A method call expression is indexable. A conditional expression is indexable if its true and false expressions are indexable. All other expressions are not indexable.

Invalid select expressions

The subject of a field selection expression must be a selectable expression. A variable expression is selectable. A concatenation expression is selectable. An array construction expression is selectable. A slicing expression is selectable. An indexing expression is selectable. A select expression is selectable. A method call expression is selectable. A conditional expression is selectable if its true and false expressions are selectable. All other expressions are not selectable.

4.3.2 Resolution

There are three kinds of resolution that link identifiers to declarations. The first kind of resulotion is method resolution that links method call expressions to corresponding symbol declarations: symbol definitions or imported external symbol declarations. The second kind of resolution is variable resolution that links variable expressions to corresponding variable declarations: formal parameters or local variable declarations. The third kind of resolution is label resolution that links identifiers of control flow statements to labeled statements.

Imports

A protocol description can import zero or more qualified identifiers. Each qualified identifier is either *registered*, or is a *standard import*. Below, we list standard imports and what symbol declarations are imported.

- `std.reo` — standard Reo connectors:
  component `sync(in, out)`
  component `syncdrain(in, in)`
  component `syncspout(out, out)`
  component `asyncdrain(in, in)`
  component `asyncspout(out, out)`
  component `merger(in[], out)`
  component `router(in, out[])`

component `consensus(in[], out)`
component `replicator(in, out[])`
component `alternator(in[], out)`
component `roundrobin(in, out[])`
component `reonode(in[], out[])`
component `fifo(in, out)`
component `xfifo(in, out, msg)`
component `nfifo(in, out, int)`
component `ufifo(in, out)`

- `std.buf` — functions for manipulating message buffers in network byte-order:
  function `byte writeByte(msg, short, byte)`
  function `short writeShort(msg, short, short)`
  function `int writeInt(msg, short, int)`
  function `long writeLong(msg, short, long)`
  function `byte readByte(msg, short)`
  function `short readShort(msg, short)`
  function `int readInt(msg, short)`
  function `long readLong(msg, short)`

Method resolution

Occurrences of imports and symbol definitions result in a list of declarations per protocol description. Each method expressions that occurs anywhere in a protocol description must refer to a listed symbol declaration. Otherwise, it is unclear what the method refers to. The process of checking whether every method expression refers to a known declaration is called method resolution. It proceeds by listing external and source declarations of composite, primitive and function definitions. After listing all declarations, method expressions are linked to the external or source declarations they refer to.

Every symbol declaration consists of a signature. A signature consists of a list of type annotations, one for each corresponding parameter. A signature is either a component signature or a function signature. A function signature furthermore has a return type annotation.

Uniqueness of symbol declarations

Every listed declaration, being a source declaration resulting from a symbol definition or an external declaration from an import, must have a unique identifier.

Valid method call occurrences

Every method call expression must refer to a symbol which is declared. The result of resolving methods is that each such expression is linked to the corresponding symbol declaration. Within any symbol definition, no method call expression must refer to a component declaration, except for method call expressions that occur in a new statement. Moreover, new statements must only occur in composite definitions, and its method call expression must refer to a component declaration.

Variable resolution

In component and function definitions, formal parameters are declared. In block statements, local variables are declared. In synchronous statements, formal parameters are declared. Each variable expression must refer to a variable declaration or a formal parameter that is in *variable scope*. Otherwise, it is unclear what the variable refers to. The process of checking whether every variable refers to a variable declaration is called variable resolution. The first step in variable resolution is to link every scope to its parent scope. Component and function definitions have no parent scope. A variable is in scope whenever it is declared in any of its surrounding block statements, or occurs as a formal parameter of a surrounding synchronous statement or symbol definition. It proceeds by linking every variable occurrence to a variable declaration in one of its parent scopes.

Block statements can be nested to declare additional local variables. Only statements that occur within a block can refer to its declared local variables. Outside of the block, those variable declarations are inaccessible because they are out of scope. It is possible to have two sibling block statements that declare the same local variable; these are different variable declarations. In other words, the scope of a block statement is restricted to its nested statements only.

Similar for synchronous statements: only statements that occur within a synchronous statement can refer to its formal parameters. Two sibling synchronous statements may declare the same formal parameter; these are different variable declarations.

Uniqueness of variable declarations

In every scope, every variable declaration must have a unique identifier. This implies that every formal parameter must be unique. Every local variable declaration in a block statement must be unique. Every formal parameter of a synchronous statement must be unique. Moreover, every variable declaration cannot overshadow identifiers already declared before by a formal parameter or a local variable.

```
composite main(in a, out a) { // illegal
    new lossysync(a, a);
}
composite main2(in a, out b) {
    channel a -> c; // illegal
    new lossysync(a, b);
}
primitive lossysync(in a, out b) {
    while (true) {
        synchronous (int a) { // illegal
            if (fires(a) && fires(b)) {
                msg x = get(a);
                put(b, x);
            } else if (fires(a)) {
                msg x = get(a);
            } else assert !fires(b);
}   }   }
```

In above example, formal parameters with the same identifier are illegal. Variable declarations that overshadow a variable already in scope are illegal. Sibling blocks that declare a

variable with the same identifier, as seen in the two cases where both `msg x` is declared, is allowed. Although the identifier is the same, the two separate blocks declare a *different* variable.

Label resolution

Occurrences of labeled statements result in a list of labels per block statement. Similar to variable resolutions, labeled statements are bound to their *label scope*. The scope of the label of a labeled statement is its surrounding block statement. Each control flow disruption statement that occurs in a symbol definition must refer to a label that is in scope. The process of checking whether every disruptive statement refers to a label is called label resolution. We already assume that block statements are linked to a parent variable scope. Thus, for every statement, it is possible to find its surrounding block statements. We collect the labeled statements that occur within a block statement.

Uniqueness of labels

In every label scope, every label must have a unique identifier. It is possible for two sibling blocks to contain labeled statements with the same identifier, but these labels are considered different.

```
int main() {
    while (true) {
        dupl: skip;
    }
    dupl: goto dupl;
}
```

In above example the duplicated label is illegal.

Valid control flow disruption statements

A goto statement must refer to a label in scope. A break or continue statement has an optional label. If a label is provided, that label must be in scope and attached directly to a while statement. If no label is provided, there must be a surrounding while statement.

Bibliography

[DA18] Kasper Dokter and Farhad Arbab. Treo: Textual syntax for reo connectors. *arXiv preprint arXiv:1806.09852*, 2018.

[ISO18] Information technology – Programming languages – C. Standard, International Organization for Standardization, June 2018.

[JSGB00] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. The java language specification, 2000.

Chapter 5

Protocol State Representation

This chapter gives meaning to protocol descriptions, whose syntax is detailed in Chapter 4. Section 5.1 introduces the table model for executions in discrete, synchronous time steps. Section 5.2 provides an informal semantics of PDL, and relates it to an ideal definition of protocol execution as a set of infinite tables. Section 5.3 introduces semantics for the execution of a connector, relating it to that of protocols, and ultimately defining a correctness criterion to serve as the foundation for evaluating the correctness of connector implementations.

5.1 The Table Model

We introduce the table model for reasoning about stateful execution in PDL components and connectors. First and foremost, a table is a structure with a set of columns, each with a unique label, and a set of rows, labeled with a whole number. Every combination of column and row intersects at a table cell, which holds a value.

We say a table corresponds to the run through a primitive component if it wholly represents the primitive's values in an observation that can be explained by a trace through its control flow. The primitive's local variables (including port variables) are reflected in columns, and the rows reflect observations of these values in agreement with its definition, one synchronous round at a time at the end of a synchronous block (see Chapter 4). The semantics of PDL expressions, statements, etc., are provided more concretely in Section 5.2 to follow. One such table runs from its initial state in the first row, advancing downward through table rows one at a time. In a run whose primitive's control flow reaches the end of the component's definition, the table propagates the values of local variables and blocks all its ports (they have the no-message value) for synchronous rounds forevermore.

The table of a composite component is defined as the combination of the tables of its constituent components, with columns renamed as necessary to avoid unintentional overlap in variable or port names. As reflected in PDL, composite components are able to fuse a pair of its constituent components' ports (one input and one output port) by creating a channel between them. This fusion is reflected in the composed table if, for every row, the fused columns have equivalent values. Often, we refer to this as 'overlapping' the ports' columns, and imagine them as a single, new column. By construction, rows of constituent components are kept 'in sync' by being mapped by the same whole number. The restrictions of the control flow of all constituent primitives, complete with local variables, are retained in the composite table.

Conceptually, a protocol description denotes a set of *infinite tables*, each denoting a unique execution of the protocol component where there are infinite rows and infinite columns. In practice, we often reason about particular runs, or finite prefixes of an infinite table. For example, a practical definition of termination is that a given infinite table has only inactivity (silent port behavior) in all rows after a finite prefix of activity (non-silent port behavior).

5.2 Protocol Semantics

We shall use the framework of denotational semantics for giving a compositional semantics of protocol descriptions. The semantics of composite and primitive component definition differ: the former is declarative, the latter is imperative. The definition of the semantics of components is given last.

In giving this semantics, we shall assume there are no divergences: the programmer is assumed to be an expert to ensure that all function definitions and synchronous statements are terminating. This, obviously, simplifies the presentation of the semantics. In practice, one would likely count the resources such as space and time consumed, and halt after a certain threshold has been reached (out of memory, out of time).

5.2.1 Internal Language

We shall restrict ourselves to a subset of the protocol description language. This restricted subset is called the *internal language*. Conceptually, one can translate the full protocol description language into this restricted subset, without losing any of the intended meaning. In fact, the meaning of the full protocol description language is precisely the meaning of its image under this translation in the semantic definition that follows.

The purpose of the internal language is to allow for a simpler semantic definition than when working with the full language. The internal language has the following properties:

- Modified assignment statements, e.g. `x += 5`, are translated into assignment expressions, e.g. `x = x + 5`.
- Side-effectful expressions are assignments and calls to built-ins. All other expressions are side-effect free if they do not contain any nested side-effectful expression. In the internal language, there are two different expressions occurring in expression statements: assignment of a built-in and assignment of a side-effect free expression. Expression statements that involve only a built-in call or a side-effect free expression can be assigned to a dummy local variable.
- Statements such as if, while, return and assert have side-effect free expressions.
- During translation, local variable declarations are introduced where necessary, using fresh local variable identifiers that do not clash with other identifiers in scope. For example, the statement `x = y = 5;` is translated into two statements `y = 5; x = y`.
- Statements consisting of disruptive control flow statements are translated into a statement where no disruptive control flow statement occurs. That is, no `break`, `continue` or `goto` statements appear, and there are no labeled statements.
- The block statement directly under the symbol definition is the only one to have non-zero number of local variables declarations. All nested declarations are transformed by moving them up in scope, possibly renaming the identifiers if clashes would occur.

We shall, for now, leave out the details of how this translation is defined. The reader may consult existing literature on the topic for a more detailed discussion. More importantly, the translation of the full protocol description language into the internal language must preserve its well-typedness.

5.2.2 Denotation

Values of the Boolean domain are true and false. We assume standard Boolean operations. Just like in Java, the primitive types *byte*, *short*, *int*, and *long*, are signed integers of finite bit width of 8-bit, 16-bit, 32-bit, and 64-bit, respectively. Bitwise, logical and arithmetical operations on these domains are defined as usual.

Arrays of non-array values are values, i.e. there are no arrays of arrays. The domain of the elements of an array is uniformly fixed, i.e. we have byte arrays and int arrays, but not arrays of mixed bytes and integers. Every array has a fixed length, of the 32-bit signed integer domain. The domain of byte arrays is included in the domain of messages, i.e. every message can be treated as a byte array. Moreover, the domain of messages includes the *null* message, representing the absence of a message.

A frame is an assignment of formal parameters (of a function declaration or the component declaration that are not of input or output type) to values. Frames are treated in such way to remain constant for the duration of a function or a component, and in particular are not updated. Moreover, for each function definition, we assume there is a corresponding function, that maps frames to return values.

A *store* is a mapping from memory variables to values. As usual, stores can be updated to overwrite the value of a variable. An *oracle* is a mapping of port variables to a value of the message domain. We say that a port fires (with respect to an oracle or the head of an infinite sequence of oracles) iff its value is not the *null* message, i.e. a message is present.

For giving the semantics of a component, we assume we are given an infinite sequence of oracles. Conceptually, every time we synchronize an oracle is taken from the stream. This oracle is used to give the values of port variables, for the duration of synchronization. Before, in between, and after synchronization, no oracles are used and port variables are inaccessible (see Section 4.3.1).

During execution of a component, we record an input/output history. This history is a sequence of events. An event is either a *get* event, a *put* event, a *tick* event, or an *inconsistency* event. The purpose of keeping track of a history is to determine the consistency of an execution. A get event records a port and a message value that is received; a put event records a port and message value that is sent. A tick event records that the synchronous round finishes, and an inconsistency event records an impossible execution.

We consider the state of a component to be represented by a triplet of some store, an infinite sequence of oracles, and an input/output history. The store is variant and represents the *current values* of the local variables and parameters. The head of the infinite sequence of oracles represents the *current values* of the *ports*, i.e. the parameters of input or output type. The tail of the infinite sequence of oracles represent all *future* values of ports. Since time progresses one step after completing a synchronous statement, the infinite sequence of oracles is also variant. The input/output history tracks the *past* events.

We start at the lowest level of the abstract syntax tree and work upwards: first we give meaning to side-effect free expressions, secondly to statements, and thirdly to the primitive component definition.

Expressions

As usual, the meaning of an expression is obtained through evaluation. Evaluation of an expression takes a state. Evaluating an expression either results in a value, a divergence, or an inconsistency. Evaluation is defined inductively on the structure of expressions, as usual. For example, a variable expression denotes the value of that variable in the store. The built-in methods of *get*, *avail* and *ready*, however, denotes the value that the head oracle assigns to the port supplied as argument to these methods. Only the *put* built-in method can cause an inconsistency during evaluation, namely when the value assigned by the head oracle and the value denoted by its second argument expression are different.

Other expressions, such as binary expressions and unary expressions, are defined in the usual way. Without being too precise, we intend to mimic the semantics of C for evaluation of expressions. Method call expressions in primitive definitions always refer to function declarations; we evaluate a function call by creating a frame out of the values obtained from evaluating the arguments and applying it to the denotation of the function definition to obtain the resulting value or a divergence. Other than function calls, in no way can expressions in primitive definitions lead to divergence.

Statements

The meaning of a statement is a state transformer, viz. a mapping from state to state. The meaning of statements is defined inductively in a standard way. For example, sequential composition of two statements is function composition of the state transformers of the comprised statements. We make the following remarks:

- The meaning of a synchronous statement is defined as follows: take the state transformer of the nested statement, and take a state. We apply the state to the given state transformer to obtain another state. In the latter state, we replace the infinite stream of oracles by its tail. This ensures that after a synchronous statement completes, we progress to the next oracle. We further record in the input/output history a tick.
- A state is inconsistent if its history has a recorded inconsistency event. The meaning of statements on inconsistent states is undefined, i.e. any further next state is related to an inconsistent state.
- For an expression statement, there are two cases: an assignment of a side-effect free expression or an assignment of a built-in call. In the former, the side-effect free expression never cause an inconsistency, and the evaluated value of the expression is used to update the store. In the latter case, encountering a built-in call, we additionally record the call and its argument values in the input/output history. If the expression evaluates to an inconsistency, we record an inconsistency event (thus making the resulting state inconsistent).
- For an assert statement, if the evaluation of the test expression is false, an inconsistency event is generated.

We consider operational consistency and causal consistency of histories. Operational consistency is the lack of any inconsistency events. Causal consistency of a history means that messages being send and received correspond to each other, i.e. there is no received message that was never sent.

Operationally Consistent Executions

An execution of a primitive component starts with some initial state and proceeds by following the state transformer that the main block statement induces. During this process, the infinite sequence of oracles is consumed, and a history of input/output events generated. The resulting execution is operationally consistent if (1) the history of input/output events does not contain an inconsistency event, (2) in each round where an input port fires according to the oracle a corresponding get event is recorded, and (3) in each round where an output port fires, a corresponding put event is recorded.

In other words, if an input port fires, there must be at least one get operation in the same synchronous round. If an output port fires, there must be at least one put operation in the same synchronous round. And as a consequence of composition, when two primitive components share a port, such that one has access to the input side and the other has access to the output side, if the port fires then at least one put and at least one get operation must be recorded in the input/output history of the composed component.

Causally Consistent Executions

Causal dependency is a relation defined between component states. Within the execution of a component, every state transition is caused by a statement. We informally consider the causal dependency relation, by looking at how statements are causally related. A statement x is causally related to another statement y in the same primitive if x has a value which y observes. For example, `x=5;y=x;` where $y = x$ is causally related to $x = 5$. A statement containing a `get` expression of the value of a port P is causally related to all put statements of port P occurring within any synchronous blocks occurring in the same synchronous round. Note that this also introduces causal relations between primitives. The causal relation is transitively closed. Executions are causally consistent if for each synchronous round the dependency graph is acyclic. This captures the intuitive causal 'flow' of data from putters to getters; we prohibit protocol descriptions where the value of a message is caused by itself. Preserving causal consistency has a desirable consequence: any given message can be traced to definitive origins, which must be either (1) being put into the protocol at a boundary port, or (2) a message valuation provided by an oracle.

5.2.3 Protocol Behavior

The ideal meaning of a protocol is a set of infinitely long tables. This set characterizes the behavior of a protocol. For each protocol description, we interpret it as describing such a set. The meaning of a protocol description is the set of infinite tables induced by consistent executions of its main component. A consistent execution of a component is an execution that is both operationally consistent and causally consistent. An infinite sequence of oracles can be treated as an infinite table, where each oracle provides the valuation of the ports of the table. A set of consistent executions induces a set of infinite tables as follows: take for each consistent execution the infinite sequence of oracles and collect them in the induced set of infinite tables.

A protocol is consistent if it consists of at least one infinite table, i.e., there exists at least one consistent execution.

5.3 Connector Semantics

This section defines the semantics of connectors. Intuitively, connectors instantiate a particular communication session with respect to a configured protocol. As such, the semantics of protocols and connectors are tightly related.

5.3.1 Setup Phase

In Chapters 2 and 3, it was explained that connectors are configured with a protocol description, that describes a set of executions. Ports of connectors are then bound, by either giving applications access to native ports, or passively or actively bind them over the Internet. At connection time, a set of peers is gathered, until the connector is fully connected.

During connection time connectors exchange configured protocol descriptions, composing the protocol descriptions. In the end, the connector consists of a single, shared protocol description that is composed out of the protocol descriptions submitted locally by each application. After the connector is fully connected, no further peers can join. Thus the meaning of the shared protocol description becomes fixed, and the communication phase can begin.

There is an important distinction between no protocol behavior, and silent protocol behavior. A connector has no protocol behavior if always eventually runs into an inconsistent execution. An inconsistent execution is not continued; there does not exists a next synchronous round after encountering an inconsistency. However, silent protocol behavior indicates that ports are always silent: further synchronous rounds can be constructed but every port blocks. Idealistically, silent protocol behavior consists of a single infinite table where all ports block, whereas no protocol behavior is an empty set.

5.3.2 Communication Phase

A connector runs through connector states while collecting constraints from native applications. A run is a sequence of connector states alternated by a constraint. The constraint is provided by the participating applications. A connector state consists of a finite table it has constructed so far.

The only permissible runs of a connector are those where its table is a prefix of one infinite table in the behavior of its associated protocol. In other words, a run consists of a finite table that must be a prefix of some infinite table specified by the composed protocol description. The infinite table is an ideal object, and the run is its finite approximation. As a connector runs through an execution, it maintains a state and records its past observations reflecting the run. Sometimes a choice can be made in which execution to follow through: two executions that are allowed by the protocol might share a common prefix with the current run of the connector.

Connectors are also restricted by native applications: its choices must not only be consistent with some behavior of the protocol, it must also be consistent with the constraints put forward by the native applications. Native applications submit constraints using the Application Programming Interface, which has an effect on the run. In Chapter 3 we further explained how the application does this, and how it is related to data flowing in and out of the connector.

An application's *mode of operation* with its connector defines how the connector organizes the application's submitted batches into synchronous rounds.

1. In *Cooperative Mode*, the connector guarantees a one-to-one correspondence between an application's `sync` invocations and synchronous rounds. The connector cannot proceed to the next synchronous round without the cooperation of the application.

2. In *Preemptive Mode*, the connector imposes *delay insensitivity* on the application; the application's constraints per `sync` are applied to synchronous rounds in the same order as in cooperative mode, but the connector is permitted to inject rounds in-between in which it presumes the application blocks all its ports. This mode can be viewed as an optimization, allowing the connector to progress its state if the application is slow. However, the application loses the ability to reason about the current round the connector is in.

In this work, we presume all connectors are in cooperative mode. This is motivated by it being 'conservative', with fewer possible executions for the given behavior. In fact, an implementation may also not implement preemtive mode. Observe that an application in cooperative mode is able to simulate one in preemptive mode by calling `sync` with empty an batch, i.e. a batch where all native ports block, and retrying to synchronize if the empty batch was selected. This, however, may starve the application of control.

5.3.3 Correctness

We distinguish two notions of connector correctness: total correctness and partial correctness. Total correctness of a connector means that the finite table of the last state of every run of a connector must be a prefix of a consistent execution of its corresponding protocol. Partial correctness of a connector means that the finite table of the last state of every run must be a consistent prefix of an execution of its corresponding protocol.

Chapter 6

Implementation

This chapter details the implementation of connectors, delving into their internals to explain how they implement the API defined in Chapter 3. The design of the implementation revolves around facilitating the communication phase, which realize the essential distributed communication behavior that is their purpose. As such, the chapter begins with Section 6.1, explaining how this communication is realized, taking the initialization and setup of the connectors for granted. Section 6.2 follows up with a detailed look at the in-memory representation of connector structures, and how this data is initialized and setup during the setup phase.

6.1 Synchronization Procedure

This section explains how connectors realize the communication that users expect when calling `connector_sync`, defined in Section 3.1.3 as realizing the user's prepared message exchange operations. During this procedure, control flow is temporarily diverted to the management of the connector's internals, where it coordinates with other connectors in the session in a distributed effort to find an *interaction* between the session's components which satisfies the configured protocol. Control flow is returned once the state of the session has been updated to reflect the interaction's completion.

Even a naïve implementation of the synchronization procedure is complicated, as it necessitates distributed consensus, and abstract reasoning about protocol behavior. As such, it is necessary for substantive explanation of the implementation to be both digestible, but not to omit important details. A balance is struck by its explanation being provided in two sections, where Section 6.1.1 provides a 'top-down' explanation, prioritizing brevity. Section 6.1.2 provides the complementary 'bottom-up' explanation, prioritizing the thorough inclusion of examples, and motivation for the design decisions made.

6.1.1 Overview

When seen as a system whose distributed connectors cooperate toward a coherent distributed task, the distributed connector runtime works each round to identify, and reach consensus on the *interaction* that characterizes the synchronous message exchanges between components. Once found, the session's state is updated, such that components' local *actions* are consistent with the interaction.

In a nutshell, the procedure works through the cooperation of two inter-related systems, both of which involve tasks distributed over the session's connectors:

1. **Speculative component execution**
   The session's components are speculative executed, exploring the possible outcomes of the synchronous round, i.e., incrementally exploring the set of possible interactions.

2. **Consensus procedure**
   The set of distributed connectors exchange and aggregate control information about their components' progress toward realizing an interaction until the designated *leader* connector *decides* in favour of the first interaction is concludes satisfies the session's protocol, whereupon the announcement of the decision is disseminated to the other connectors.

Figure 6.1 gives a graphical representation an example session, where connectors and components are shown as vertices, and their relationships driving procedures (1) and (2) are shown with solid and dashed arrows respectively.

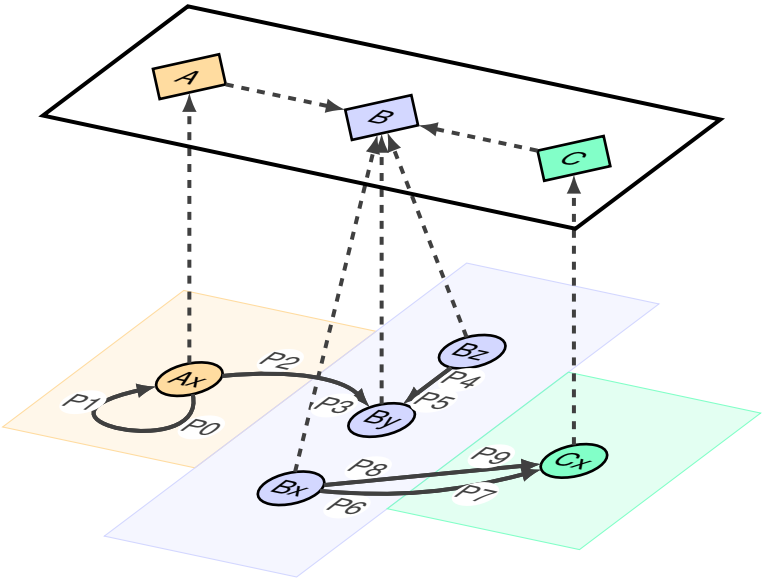


Figure 6.1: An example session with three connectors (diamonds), partitioning the management of a set of five components (circles), with the positioning, color, and naming of components agreeing with that of their respective manager. Dashed arrows show edges of the solution tree, from child to parent. Solid arrows show logical message channels between components (with ports on either end), facilitating the transmission of speculative messages, such that the speculative executions of the sender component can inform that of the recipient component.

6.1.2 Incremental Explanation

This section lays out a feasible design process behind the final implementation of the synchronization procedure, such that ultimately, we arrive at a design consistent with the overview in Section 6.1.1 above, and with its example session shown in Figure 6.1.

Before immediately delving into the specifics which come together to realize the implementation of `connector_sync` in its final form, it is helpful to clarify the problem the distributed connector runtime must solve. As it was described in Chapter 5, Reowolf reasons about the behavior of a session being a sequence of *rounds*, in which each of the session's ports exchange up to one message. It is accurate to think of the task of the distributed connector runtime as, each round, solving an instance of the *distributed constraint satisfaction* ('DCS') problem [YDIK98], where (a) the solution is the assignment of an (optional) message value to every port, and (b) the session's protocol provides a constraint on the set of satisfactory solutions. With this in mind, we henceforth consider terms 'interaction' and 'solution' to be synonymous.

The remainder of the section describes the implementation's approach to solving these DCS problems each round. The solution is best understood by approaching it incrementally, starting from the most naïve solution, which is easy to understand, but whose other properties make it infeasible in practice.

Exhaustive Solution Search

To begin, we outline the most essential approach to solving constraint satisfaction problems, achieving simplicity by (a) reasoning at a monolithic level, i.e., not distributed, (b) assuming we have a means of checking whether a given candidate satisfies the protocol, and (c) completely disregarding performance characteristics such as efficient utilization of time and network bandwidth.

With these simplifications, we may envision a conceptually simple, and functional implementation: the runtime enumerates all conceivable *solution candidates* in arbitrary order, and selects one to be the solution if it satisfies every components' constraints. Given that our message domain is finite (i.e., messages have a maximum permitted length), and the number of ports is finite, then there is always a finite solution space to search. Ergo, this approach eventually identifies a satisfactory solution, if it exists.

To clarify this description by way of an example, consider the constraint satisfaction problem which corresponds to identifying a satisfactory interaction for some round of a session with two ports, $\{p, g\}$. The solver enumerates candidates of the form $\{p \mapsto M_p, g \mapsto M_g\}$, where M_p and M_g are port values. After considering, and rejecting some 65 failed candidates, the candidate where $M_p = M_g = [22, 22]$ is found to satisfy the protocol, whereupon it is selected as the solution. The round ends with all components' states updated to reflect the completion of the round where ports p and g exchange the two-byte message, $[22, 22]$. Henceforth in this section, we represent message values using sequences of hexadecimal-encoded bytes in this fashion, and use $*$ to represent the special port value that is the absence of a message, distinct from the zero-length message, [ ].

Distributing the Solution Search

Our algorithm is still far from feasible, however. The most obvious problem is that it isn't suited for a distributed setting, in which every connector has only a partial view of the constraints; e.g. applications express their requirements in terms of synchronous batches to their local connector (see Section B.2.2). To proceed, we distribute connectors over the network, such that each has its own local memory, and can only communicate with its peers through the explicit exchange of control messages over the network. Furthermore, we distribute the set of components over the set of connectors, assigning one *manager* for each; as such, every connector manages a set of components.

Adapting our algorithm for use in our distributed system requires addressing two resulting complications:

1. How can we check whether a given candidate solution satisfies the protocol's constraints, when it is defined in terms of components distributed over the set of components?

2. In cases where the candidate space contains multiple solutions, how do we ensure that the connectors reach consensus on which solution is selected as *the* solution?

Toward solving problem (1), recall that Reowolf's protocols are compositional, with the behavioral constraints of composite components expressed in terms of the composition of their constituent component's constraints. We lean on this compositionality to enable the efforts of checking whether candidate solution satisfies the session's protocol, despite the session's components being distributed over the connectors' memory: a candidate solution satisfies the solution's protocol if and only if it satisfies the constraints contributed by all components. In other words, the compositionality property of PDL allows us to reason about the session's constraints on satisfactory solutions piece-wise, with one piece per component.

Toward solving problem (2), we introduce the *consensus tree*, a network (specifically, a tree) overlaid on that provided to connectors already in the form of transport-layer channels between connectors; concretely, the consensus tree is a directed, acyclic graph, whose nodes are connectors, and whose edges are a subset of the edges corresponding to transport channels between the connectors. The imposition of this tree structure breaks the symmetry between the connectors, simplifying their cooperation on tasks that require making choices. The root node of the consensus tree, in particular, is assigned a special role, and is thus given a special name: the *leader*[1]. Finally, we establish local tasks for connectors to perform, specialized to their situation in the consensus tree, allowing a coherent, distributed task to emerge from the coordinated efforts of the connectors, which we call the *consensus procedure*: *suggestion* messages travel 'down' the tree (toward the leader), each encoding a candidate solution, x, whenever x satisfies the constraints of all components managed by all connectors in the solution *sub-tree* rooted at the child. What emerges is a 'filtering' of candidates down the tree; at every level, connectors work to inform their parents of candidates that they have found to satisfy all components in the sub-tree of which they are the root. Consensus with this approach is achieved easily. Only the leader is permitted to decide on any solution in particular; as long the leader takes care to do so only once per round, the round's decision event is unique. All non-leader connectors are informed of the decision from their parent, in the form of an *announcement* message. Note that edges in the consensus tree carry suggestion and announcement messages in fixed, opposite directions, for the entirety of the session.

To demonstrate our distributed algorithm, Figure 6.2 exemplifies a session of connectors A, B, and C, managing sets of components $\{Ax\}$, $\{Bx, By, Bz\}$, and $\{Cx\}$ respectively. Once B has received some candidate solution m from A, it knows that all components managed by connectors in the sub-tree rooted at A would be satisfied with m as a solution, without B even having to know what those sets of components and connectors are. Once B identifies some candidate n which satisfies all its managed components, and has been received in control messages from both A and C, n is a solution, eligible for the leader's decision.

[1] The leader is named to reflect its selection through an *election algorithm* [Fok13]; see Section 6.2.3 for more details.

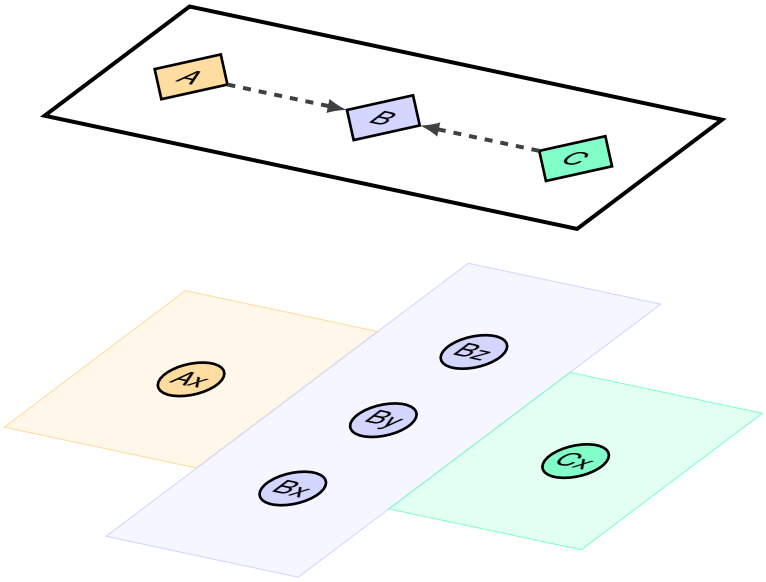


Figure 6.2: An example session with three connectors (diamonds), partitioning the management of a set of five components (circles), with the positioning, color, and naming of components agreeing with that of their respective manager. Edges of the consensus tree are shown as dashed arrows from child to parent.

Candidate predicates

Our distributed algorithm is functional, but suffers from an incredibly large candidate space; in some cases, the solution would only be found after exchanging and comparing an indefeasibly large set of rejected candidates.

Our next modification to the algorithm is an optimization that allows connectors to reason and communicate about candidates in aggregate. We introduce the *candidate predicate*, a structure which represents a set of candidate solutions. By taking advantage of the combinatorial nature of our solution space, we design a predicate encoding that predicates over the values of port values separately. Concretely, each predicate is stored as a *partial map* of port variables to port values; missing mapping effectively abstract over all possible port values. For example, for a session with port set $\{p, q\}$, the predicate $\{p \mapsto [00, 5e, ff]\}$ includes all candidates with the particular message value $[00, 5e, ff]$ for port p, but for any value for q. To take advantage of such an encoding, it is fruitful for connectors to structure their search of the candidate space, such that highly generic predicates can be aggregated and transmitted over the network. As a trivial example, consider the case of a session with connectors and components arranged as shown in Figure 6.2, previously, where every candidate is a solution; A would only need to send a single, extremely terse control message to B: the trivial predicate, $\{\}$.

Component Speculation

Until now, our approach has glossed over two important steps: (1) how precisely we check whether a given component is satisfied by a given solution, and (2) how a connector structures its exploration of the candidate space.

We clarify our algorithm by introducing a structured means of computing, for every component, the set of all candidate predicates which the component would accept. To preserve the locality of components, we imagine this task is performed by the component's manager connector. Accordingly, we introduce the *solution tree*, a super-tree of the consensus tree, which captures the flow of candidate solutions to the leader from start to finish; leaf nodes correspond to components, which enumerate the set of candidate predicates they accept, to their parent (their manager). All internal nodes of the solution tree are connectors, which work to aggregate the predicates received from their children, updating their parent (if they exist) with control messages. Figure 6.3 shows the solution tree for our running example session; observe how all leaves are components, all internal nodes are connectors, and *B* is still the leader.

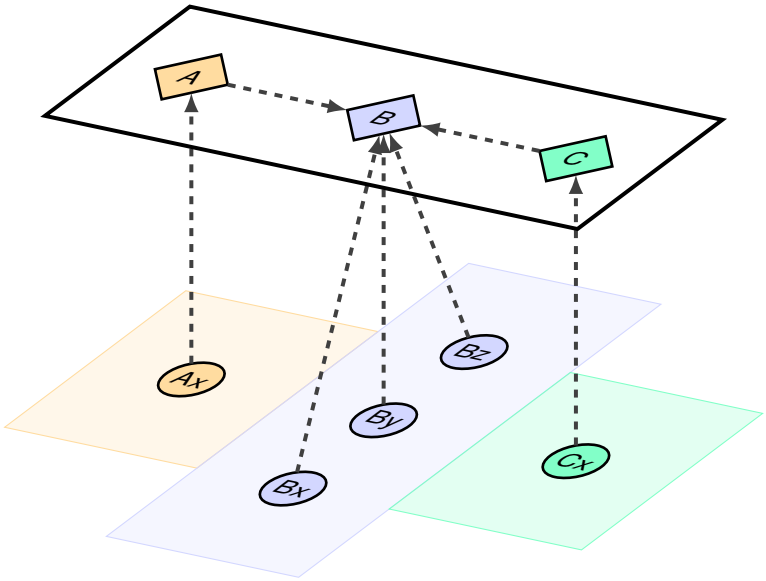


Figure 6.3: An example session with three connectors (diamonds), partitioning the management of a set of five components (circles), with the positioning, color, and naming of components agreeing with that of their respective manager. Edges of the solution tree are shown as dashed arrows from child to parent; note that leaves are components, and internal nodes are connectors.

In the case of *native components* (i.e., components corresponding to the user's application), the connector has no insight into the component's behavior. Instead, the component's requirements on the session's behavior during the round are provided explicitly, a priori, stored in the connector as *batches* (see Section B.2.2). As such, the set of predicates accepted by the native component are simply computed eagerly, at the start of the round.

The interesting case is for *protocol components*, components whose states are managed by the connector itself, and whose deterministic behavior as a function of the runtime environment is explicitly defined as part of their configuration a priori. For these components, the connector drives the exploration of its satisfactory predicates through the *speculative execution* of the protocol component in a safely encapsulated environment, forking the search with branches in control flow, and computing the candidate predicate resulting from every control flow path that reaches (speculatively) the end of the component's next *synchronous block* (see Section 5.1.

This approach complements the structured representation of candidate predicates, as protocol components are only able to express constraints on their ports' values one port at a time, allowing for the incremental refinement of candidate predicates, one port-variable-mapping at a time.

For a simple example, consider some protocol component `token_spout`, which offers to send a zero-length 'token' message out through its only port, p. Figure 6.4 shows how this component ends up submitting two candidates into the solution tree: $\{p \mapsto *\}$, and $\{p \mapsto [\,]\}$.

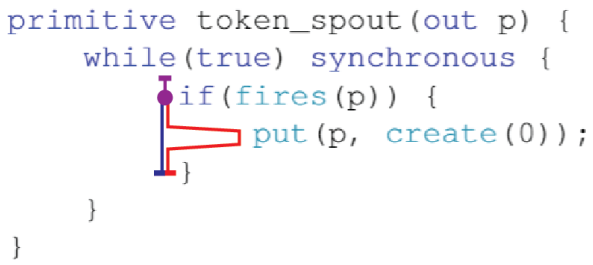


Figure 6.4: Branching, speculative execution path for a component instantiated with the `token_spout` protocol. Starting at the beginning of the `synchronous` block with the trivial predicate with speculative assignments, $\{\}$, the execution branches in response to reflection of an indeterminate speculative variable in expression `fires(p)`, enumerating the speculative values in {`false`,`true`}. The resulting two branches distinguish their behavior before both reaching the end of the `synchronous` block, whereupon they submit candidate solutions $\{p \mapsto *\}$ and $\{p \mapsto [\,]\}$.

Speculative Messaging & Lazy Speculation

The approach described thus far works well for components such as `token_stream`, whose behavior permits a very small set of candidate solutions. However, this component tends to be the exception to the rule; components tend to be particularly flexible to their environmental constraints when reasoned about in isolation. For example, consider generating the candidate set for the `msg_sink` component; it would enumerate all predicates of the form $\{i \mapsto x, o \mapsto x\}$. The definition of `msg_sink` is given in Figure 6.5.

The observation is that our current approach results in many candidates being generated in the leaves of the solution tree, only to be filtered out higher up in the tree, as a result of not being shared by other components. As a simple example, imagine the `token_stream` component defined previously, connected such that its output feeds into the input of a `msg_sink`; despite the latter generating a deluge of candidate solutions, in composition with the former, all candidates are filtered out, but those for which $g \mapsto *$ or $g \mapsto [\,]$. The current algorithm lacks a way to guide the exploration of the candidate space as a function of the available protocol information.

We modify our algorithm to delay the exploration of every new branch, speculatively assigning port value m to input-port variable x, until there has been an exploration which assigns m to output-port x', where x and x' are two ends to the same logical channel. As x and x' may belong to different components, potentially managed by altogether different connectors, this information may require the exchange of a *speculative message* over the network, between connectors. The resulting runtime behavior is surprisingly intuitive: a component calling `get` really does block until the receipt of a message from another component calling `put`. An important consequence of this change is that the runtime will not find solutions whose sessions

contain cyclic causal dependencies between message contents; Reowolf's definition of causal dependency is precisely defined in Section 5.2.2, but here it suffices to say that it captures the intuition that we are only interested in solutions where there is some causal ordering on messages sent as part of a synchronous interaction, excluding interactions where messages' contents are defined in terms of themselves. To aid with the reader's intuition, this direct communication of otherwise concurrently-executed components bears many similarities to the *actor model*, a well-understood model of concurrent computation [Hew10].

With this modification, the speculative execution of `msg_sink` blocks for control messages rather than speculating about the message received by port g. Figure 6.5 shows the branching path of the speculative execution of a round for the `msg_sink`; observe that branch that calls `get` blocks, awaiting control messages. If the component owning the counterpart to port g never sends any messages, this component never considers candidates that receive messages at g.

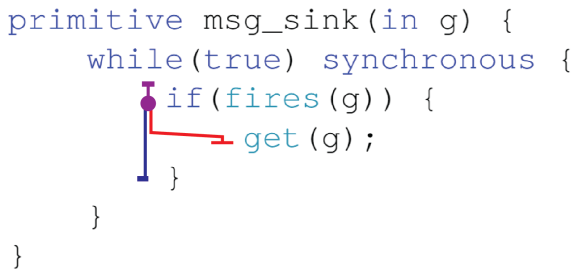


Figure 6.5: Branching, speculative execution path for a component instantiated with the `msg_sink` protocol. Starting at the beginning of the `synchronous` block with the trivial predicate with speculative assignments, $\{\}$, the execution branches in response to reflection of an indeterminate speculative variable in expression `fires(p)`, enumerating the speculative values in $\{\texttt{false}, \texttt{true}\}$. While the `false` branch reaches the end of the synchronous block, submitting candidate $\{g \mapsto *\}$, the latter branch *blocks* rather than speculating about the message received at port g.

To demonstrate two components' speculative executions cooperating in their exploration of the candidate space, consider a session comprised of components `token_stream` and `msg_sink`, connected by a single logical channel. Figure 6.6 shows the speculative execution of the latter component blocks until it receives a control message from the former component, triggering the exploration of a branch with $g \mapsto [\,]$. Ultimately, both components submit candidates, such that the round is decided in favor of the solution $\{p \mapsto [\,], g \mapsto [\,]\}$.

All together, the refined algorithm has two complementary systems working in tandem to solve the DCS problem each round: (1) the branching, speculative execution of the session's components, with speculative control messages passing speculative message information between components, submitting candidate predicates into the solution tree, and (2) the aggregation and collection of candidate predicates filtering down the solution tree toward the leader, who chooses the first predicate found to satisfy all components as the round's solution. Figure 6.1 gives an updated illustration of our running example session.

Symbolic Predicates

Our final modification addresses the representation of candidate predicates. As they are currently described, they encode a partial mapping from port variables to port values, such that

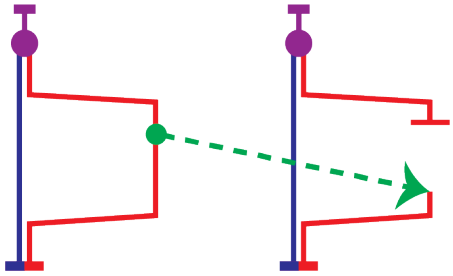


Figure 6.6: The branching speculative execution paths of components `token_spout` and `msg_sink` on the left and right, respectively, in a session where their respective ports are connected by a logical channel. The speculative execution of the latter component blocks at `get` until the receipt of a control message send by the former at `put`, triggering the continued exploration of the blocked branch with the provided message value. Ultimately, each component has two branches reaching the end of the synchronous block, and submitting a candidate predicate.

an example of a predicate for a session with ports $\{a, b, c, d\}$ is $\{a \mapsto *, c \mapsto [e2, f2], d \mapsto [\,]\}$. The observation is that this encoding of predicates results in frequently-performed operations' costs scaling with the size of port values (which can be vary large indeed). For example, as part of the consensus procedure, predicate $p = \{a \mapsto m\}$, where m is some 1000-byte message, the repeated transmission of p down the solution tree, and repeated comparison of p to other predicates may incur significant overhead as a consequence of the size of its representation.

To remedy this problem, we modify candidates and predicates to reason about solutions to a *symbolic* DCS problem, solving the assignment of *speculative variables* to *speculative values* rather than port variables to port values. To make this modification useful, we also require a correspondence between solutions to the symbolic DCS problem (decided by the leader) to solutions of the 'real' DCS problem, determining the messages exchanged at the session's ports. The idea is that it suffices for symbolic solutions to define, for each of the session's protocol components, the *path* they took in their speculative execution from the start to end of their synchronous block. The assumption is that there are far fewer paths through a components' synchronous block, than there are elements in the domain of port values, and yet the former determines the latter in a given round.

Detailed example of one synchronous round

Bringing it all together, we walk through the progress of a particular synchronous round, depicted in Figure 6.7, whose components' protocols and branching speculative executions are depicted in Figures 6.4 and 6.5. The round is solved through the completion of the following three tasks, completed concurrently but for their explicitly described inter-dependencies:

- **Speculative execution of `token_spout`**
  The speculative execution of the `token_spout` completes two branches unhindered, as neither branch is causally dependent on other components' branches; ultimately, two candidate predicates are submitted into the solution tree, rendered as $\{V \mapsto 0\}$ and $\{V \mapsto 1\}$ respectively. Here, V is some speculative variable, encoding the result of `fires(p)`, i.e., 'whether port p fires, sending some message', this being sufficient to uniquely distinguish

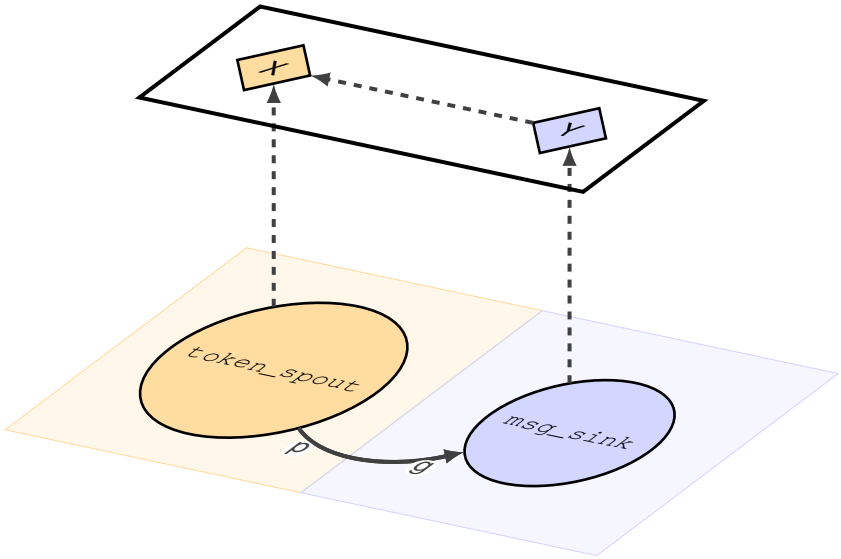


Figure 6.7: An example session with two connectors (diamonds), partitioning the management of a set of two components (circles), with `token_spout` managed by *A* and `msg_sink` managed by *B*. Edges of the solution tree are shown as dashed arrows from child to parent; note that leaves are components, and internal nodes are connectors.

the two branches; as V_p has a boolean domain, it theoretically requires only one bit[2] of space to represent its value in the predicate.

Upon reaching `put`, the latter branch transmits a speculative message, with message [] from this component to `msg_sync`, carrying with it an annotation of the candidate predicate $\{V \mapsto 1\}$, such that the latter component can recognize the speculative context in which the message is sent, i.e., which speculative assumptions were made by the branch of `token_stream` that sent the message.

- **Speculative execution of `msg_sink`**
  It, too, forks into two branches, annotated with predicates $\{V \mapsto 0\}$ and $\{V \mapsto 1\}$ respectively. The latter speculative branch of `msg_sync` blocks on `get`, aware that it awaits some message, but unwilling to speculate on the message's contents.

  At some point, the speculative message sent from the `token_stream` component arrives; by inspecting the predicate it carries, the blocked speculative branch of `msg_sync` adopts the message, unblocking `get`. Ultimately, `msg_sync` submits two candidate predicates: $\{V \mapsto 0\}$ and $\{V \mapsto 1\}$.

- **Consensus procedure**
  Candidate predicates $\{V \mapsto 0\}$ and $\{V \mapsto 1\}$ are submitted independently at the solution tree's leaf nodes, `token_spout` and `msg_sink`. *Y* determines that all of its children (just `msg_sink`) have candidates consistent with $\{V \mapsto 0\}$ and $\{V \mapsto 1\}$, and so it forwards

[2] In practice, the implementation uses variable-length integer encoding ubiquitously, representing these small integers as one byte each. Section 12.2.11 describes future work toward a denser predicate encoding.

them to its parent, X, using control messages. The two predicates race to X over the network. $\{V \mapsto 1\}$ happens to arrive and be processed by X first.

X recognizes that all its children have a predicate consistent with $\{V \mapsto 1\}$; lacking a parent, X decides this predicate to be the solution, announcing it to all child connectors, $\{Y\}$, and discontinuing the consideration of further solutions. The round ends, with X and Y consensing on $\{V \mapsto 1\}$ as the symbolic solution.

Note that the speculative execution of both components use the same speculative variable V to distinguish their branches. This is because, in this particular session, the two ports, p and g, actually share the same logical channel, and thereby, speculations of one port's value correspond with speculations of the other.[3]

As we expect, this session has two solutions: predicate $\{V \mapsto 0\}$ for the solution in which neither component sends nor receives a message, and predicate $\{V \mapsto 1\}$ for the solution where the former sends the message [] to the latter. The presence of multiple solutions is indicative of a session whose behavior is the result of a non-deterministic choice, left for the connector runtime itself to decide; the leader makes this choice by *greedily* selecting the first solution it finds, effectively prioritizing solutions that are more easily computed. The chosen solution is meaningless in isolation, but it suffices that all the session's components can interpret its meaning, such that they can recognize which of their branches contributed to the selected solution; concretely, they select their branch whose predicate, b meets the following requirement, given candidate predicate s: $b \subseteq s$. As the round ends, the effects of the round *commit*; all protocol components' states collapse down to that of the appropriate branch persistently, and control flow returns to the user having completed `connector_sync`.

6.2 Data Structure and Setup

This section gives a detailed view of the internals of the connector, by laying out the definitions of the `Connector` structure and its most important sub-structures (Section 6.2.1), and by walking through the initialization and modification of its fields throughout the setup phase (Sections 6.2.2 and 6.2.3), from the moment it is initialized, to the moment it transitions to the communication phase, where it is able to participate in the synchronization procedure, described previously in Section 6.1.

6.2.1 Persistent Connector Data

When using connectors, control flow is passed back and forth between the user's application, and the procedures defined in the connector API, detailed in Chapter 3. For there to be a continuous communication session that persists between calls, it is necessary to define a `Connector` structure, for storing the necessary, persistent state information, and encapsulating it in a form that facilitates the implementation of these procedures in a fashion that encapsulates the internal complexities safely from the user. This section lays out the type definitions which ultimately comprise the contents of `Connector`.

The implementation of the distributed connector runtime is was done in the Rust systems programming language, chosen for its focus on safety, and efficient inter-operability with the C programming language. The remainder of this section lays out the type definitions of the

[3] This approach ensures that the decided interaction always has the same values for putter- and getter-ports sharing a logical channel. An equivalent approach would use separate speculative variables for representing whether or not each port fires, and simulate the constraints of an implicit `sync` channel for every logical channel.

`Connector` and its most important sub-structures. This serves to allow a more concrete understanding of how connectors realize sessions, as is explained in Section 6.2.1 to follow.

Protocol Data

Protocols and protocol components have a one-to-many relationship in sessions; while protocol components may be instantiated repeatedly per session, and may differ from one another in state, their protocol descriptions never change. This immutability of a protocol definition throughout the session makes it practical for connectors to alias protocol definitions, enabling a one-to-many relationship between protocol configurations and connectors also. This is facilitated by the `ProtocolDescription` being exposed as its own structure, for the user to initialize, destroy, and use for configuring newly-created connectors.

The internals of the `ProtcolDescription` encodes a linked, decorated abstract syntax tree, corresponding more or less to the structure of the PDL description from which they are parsed:

```
struct ProtocolDescription {
    heap:   Heap,
    source: InputSource,
    root:   RootId,
}
struct Heap {
    protocol_descriptions: Arena<Root>,
    pragmas:               Arena<Pragma>,
    imports:               Arena<Import>,
    identifiers:           Arena<Identifier>,
    type_annotations:      Arena<TypeAnnotation>,
    variables:             Arena<Variable>,
    definitions:           Arena<Definition>,
    statements:            Arena<Statement>,
    expressions:           Arena<Expression>,
    declarations:          Arena<Declaration>,
}
```

A noticeable property is the bulk of their contents being stored by the `Heap` structure in various, typed `Arena` containers, whose elements are accessed by element IDs such as `RootId`. The use of arenas and keys enables their contents to represent cyclic graphical structures, while still being (de)serializable, such that protocol descriptions can be exchanged over the network without requiring that they be repeatedly parsed. Section 6.2.1 describes the distributed setup procedures which require the exchange of protocols over the network.

Connector Data

Below, the definitions of the `Connector` itself is provided, along those of its most important constituent sub-structures:

```
struct Connector {
    unphased: ConnectorUnphased,
    phased:   ConnectorPhased,
}
```

```
struct ConnectorUnphased {
    proto_description:   Arc<ProtocolDescription>,
    proto_components:    HashMap<ComponentId, ComponentState>,
    logger:              Box<dyn Logger>,
    ips:                 IdAndPortState,
    native_component_id: ComponentId,
}
enum ConnectorPhased {
    Setup        (Box<ConnectorSetup>),
    Communication(Box<ConnectorCommunication>),
}
struct ConnectorSetup {
    net_endpoint_setups: Vec<NetEndpointSetup>,
    udp_endpoint_setups: Vec<UdpEndpointSetup>,
}
struct ConnectorCommunication {
    round_index:      usize,
    endpoint_manager: EndpointManager,
    neighborhood:     Neighborhood,
    native_batches:   Vec<NativeBatch>,
    round_result:     Result<Option<RoundEndedNative>, SyncError>,
}
struct IdAndPortState {
    port_info:  PortInfoMap,
    id_manager: IdManager,
}
struct EndpointManager {
    poll:               mio::Poll,
    events:             mio::Events,
    delayed_messages:   Vec<(usize, Msg)>,
    undelayed_messages: Vec<(usize, Msg)>,
    net_endpoint_store: EndpointStore<NetEndpointExt>,
    udp_endpoint_store: EndpointStore<UdpEndpointExt>,
    io_byte_buffer:     IoByteBuffer,
}
struct Neighborhood {
    parent:   Option<usize>,
    children: VecSet<usize>,
}
```

The data stored by a connector roughly falls into three categories, corresponding with the OSI layers with which connectors are concerned:

1. **Application Layer 'Above'**
   Once the session has begun, the connector allows the application to play the part of a native component during the synchronization procedure. Between synchronous rounds, the application can call the connector to either (a) prepare data for the synchronous round to follow, or (b) inspect the result of the previous synchronous round. Both of these cases require the connector to store data for the application to access. These fields for these

two purposes are visible in the definition of the `ConnectorCommunication` struct, as fields `native_batches` and `round_result` respectively.

2. **Transport Layer 'Below'**
   Both in setup and communication phase, the connector must manage transport-layer resources, used for communicating control information with other connectors over the network.

   While the connector is in the setup phase, these transport resources have not yet been created, and instead, exist purely in the form of annotations, to be realized as network connections later. They are visible as all the contents of the `ConnectorSetup` structure.

   While the connector is in the communication phase, the connector manages a set of TCP endpoints, for sending and receiving control messages. To support the features described in Chapter 9, the connector also manages a set of UDP endpoints, for use in message communication accross the boundary between the connector-managed session, and the outside world. To facilitate the sending and receiving of messages in an asynchronous programming style (where the availability of network resources determines the order in which message exchanges occur), the `EndpointManager` structure stores endpoints along with several auxiliary fields. This includes `poll` and `events`, for facilitating the receipt and iteration over asynchronous I/O events, `io_byte_buffer`, a buffer for reading and writing serialized data, and other fields for buffering control messages that arrive through endpoints until the runtime is ready to handle them.

3. **Session Layer**
   The remaining fields are all concerned with functionality that facilitates the continuation of the distributed communication session. The session's configured protocol description is stored behind an atomically-refence-counted pointer in field `proto_description`, and is repeatedly accessed in a read-only manner during the synchronization procedure. The persistent states of protocol components are stored in `proto_components`, while the logical relationships between ports, other ports, endpoints, and components are managed by the `port_info` field. Safe management of ports and components is made possible by the management of *identifiers* by the `id_manager`. The `neighborhood` field realizes this connector's view on its role in the *consensus tree*, as described in Section 6.1.2, organizing its participation in various distributed control algorithms.

As is the convention in the Rust programming language, the implementation ubiquitously enforces the Resource Acquisition Is Initialization ('RAII') pattern, ensuring that this `Connector` structure, as well as all of its constituent fields are always in a valid, initialized state from the moment it is first returned as the result of `connector_new`, defined in the connector API.

This section details the internal view of the initialization and setup of the connector data structure, including all setup steps necessary to get the connector into the state where it is enters the communication phase, ready to participate in the synchronization procedure as it is described in Chapter 6.1.

6.2.2 Incremental Configuration

A connector is initialized into the setup state, already given a particular protocol description, from which it will draw the definitions of its protocol components. While in this state, the application is able to incrementally refine its configuration of the session, including the set of protocol components, and logical channels between components.

As described from the user's perspective in Section 3.1.2, this incremental setup is modeled as a game of port ownership management; the application creates new ports and adds them to the interface of their native component. These ports can be subsequently moved into the interface of newly-created protocol components. Internally, the connector is able to ensure that the session is always in a well-formed state, by tracking the ownership relation between ports and components, prohibiting attempts by the user to move ports which the native component does not own.

As new ports and components are created (and as appropriate, their identifiers returned), the connector modifies its internal state to initialize and retain the states of the associated resources, beyond the control of the user. The allocation of fresh port and component identifiers is performed by the `IdManager` structure, visible in the listing in Section 6.2.1 as one of the connector's fields, `id_manager`. It is essential to the synchronization procedure that all ports and components be provided unique identifiers, such that there is no ambiguity in the interpretation of control messages. Connectors ensure that the identifiers they allocate are unique by structuring them as the tuple, (*a*, *b*), where *a* is the identifier of the connector itself, and *b* is a unique suffix, whose allocation management is the sole responsibility of one `id_manager`. Effectively, identifiers are kept unique by leveraging the uniqueness of connectors' identifiers per session. Previous work, concisely presented by Wan Fokkink [Fok13] for a subclass of the possible network configurations of connectors, is shown to be an essential requirement of the consensus procedure necessary in such a distributed algorithm (concretely, shown in the literature as the impossibility of election in anonymous rings). The connector API affords two means of selecting connector identifiers: (a) users can provide their own identifier upon connector initialization, relying on the user to ensure a unique selection, or (b) the connector will use the system's random number source to guess a unique connector identifier, relying on the overwhelmingly small probability of two random guesses colliding in the same session, on account of the large space of available connector identifiers.

6.2.3 The Connect Procedure

Once the application invokes `connector_connect`, the connector works to finalize the session's currently-configured state. The remainder of this section describes the task of the connector within this procedure, returning only in the case of error, or with the successful creation of a session. The descriptions to follow repeatedly refer to the sub-structures of the `Connector`, as defined in Section 6.2.1.

Finalizing Logical Channels

The first step is to realize all of the sessions logical and transport channels, making the network traversible, and finalizing the last logical resources present at the session's start. The result is the initialization of the `EndpointManager`, establishing the set of transport endpoints, registering them with the poller, and receiving the first setup control messages through the newly-created endpoints to finalize the coupling of logical channels that span the network. This step results in the failure of the connect procedure if not all transport endpoints are connected successfully before the user-specified timeout duration has elapsed.

Subsequent steps depend on the `EndpointManager` to enable the connector exchanging messages with its peers in the network. Through the use of the I/O event poller and event buffer, this can be done using *asynchronous* message operations, such that send and receive operations can be completed in the order they become available using only one thread of control.

Consensus Tree Construction

Once all channels have been initialized, the task of the connector is to establish its relationship with neighboring connectors to ultimately situate itself in the *consensus tree*, as it is defined in Section 6.1.2. The result is the initialization of the connector's `Neighborhood`.

The construction of the consensus tree is a distributed task, built atop a sequence of two distributed algorithms.

First, the leader (i.e., root of the tree) is elected through the use of the *echo algorithm with extinction* [Fok13], resulting in the connector with the maximal ID being elected as the leader. In a nutshell, this algorithm works by every connector initializing a wave (a broadcast of messages which 'bounces' off the edges of the network, returning to the initiator), tagged with their own identifier, and awaiting an acknowledgement from each neighbor. Upon learning of a wave whose initiator has a larger identifier, the connector abandons their current wave, and propagates the new wave. By relying on the requirement that all connectors have unique identifiers, the result is a unique connector whose wave completes, completing the election.

Second, the tree is constructed by the leader initiating a new wave, announcing their identity to the rest of the connectors. The first time a connector receives this announcement from a neighbor, they consider the sender their parent, reply with a 'you are my parent' message, forward the announcement to all other neighbours. All that remains is to partition its non-parent neighbours into *children* and *strangers* (i.e., the channel between strangers is omitted from the consensus tree). From every non-parent, a connector awaits one message, determining whether theirs is a parent-child relationship: children reply 'you are my parent', while strangers send the announcement. In other words, strangers recognize their relationship by learning that neither considers the other their parent.

Session Optimization

At this point, the session is in a state where it is possible to consider the connect procedure complete. However, a final distributed procedure is performed, allowing connectors to modify their internal states to facilitate *session optimization*. In summary, this results in the mutation of the connector's initial configuration, affecting fields `protocol_components`, `port_info`, and `protocol_description`. The details of this procedure are provided in Section 8.1.

Bibliography

[Fok13] Wan Fokkink. *Distributed algorithms: an intuitive approach*. MIT Press, 2013.

[Hew10] Carl Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[YDIK98] Makoto Yokoo, Edmund H Durfee, Toru Ishida, and Kazuhiro Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on knowledge and data engineering*, 10(5):673–685, 1998.

Chapter 7

Deviation Detection

This chapter extends the description of the connector runtime implementation of Chapter 6, focusing on its safety properties. As formal verification of these properties is out of scope of the project, it is left for future work, and argued informally for now.

Section 7.1 describes how session is realized as network communications such that the configured protocol is always preserved, assuming that all participating processes are using the provided connector implementation. Section 7.2 follows up by exploring the consequences of erroneous connectors, or malicious processes masquerading as connectors, participating in sessions with legitimate connectors.

7.1 Session Protocol Deviation

This section explores the safety and liveness properties on which participants in sessions can rely, assuming that the session is comprised only of legitimate connectors, using the connector runtime implementation described in Chapter 6.

Section 7.1.1 explains that, given enough time and memory, and assuming the network remains intact, the runtime will eventually advance the session's state such that its protocol is preserved. Section 7.1.2 explains how applications are in control of the time they are willing to spend attempting to advance the state of the session, guaranteed by their connector through the use of a *distributed timeout*. Section 7.1.3 explains how, regardless of whether the attempt to advance the session's state is successful, the session is always in a consistent state.

7.1.1 The Protocol is Preserved

Given enough time, the task of the synchronization procedure is to identify and realize one interaction, and update the state of the session accordingly. While, the workings of this procedure are detailed in Section 6.1, we argue that the following property is preserved by the procedure: the session's behavior does not deviate from that permitted by its configured protocol. We orient our reasoning around the interaction decided upon, and subsequently realized, by the synchronization procedure.

The interaction per round is unique

For reasoning about the interaction per round to be sensible, it is necessary for the interaction to be unique. Generally, in a distributed system, it is a non-trivial matter of consensus for the distributed processes to have a coherent view on such a value. The implementation achieves this by relying on two other properties: (1) each session has one *leader*, who is unique in their ability to *decide*, and (2) the leader's decision is unique per round.

Property (1) is established during the setup phase, through the construction of a consistent *solution tree* overlay network atop the sessions connectors, whose root is the leader. Section 6.2.3 describes in more detail the specifics of the distributed *connect procedure*, which ends the setup phase only once this tree is constructed. In a nutshell, one of the first steps is for the session's connectors to reach consensus on the identity of the leader by way of a distributed election algorithm.

Property (2) is trivially enforced by the leader itself. Upon making a decision within some round with index N, the leader increments the value of a persistent local variable tracking N, and ending their synchronization procedure (after announcing their decision, cleaning up and so on). Henceforth, any decisions made by the leader will be for rounds whose index is strictly greater than N.

As only the leader of the session can decide, and the leader cannot decide repeatedly in the same round, we conclude that the decision per round is unique. Furthermore, as each decision encodes one particular interaction, the interaction per round is unique.

Only protocol-adherent interactions are decided upon

Next, we argue that the connector runtime only makes 'good' decisions, i.e., the interaction associated with a given decision will result in behavior that does not deviate from the session's protocol. Ultimately, it suffices for the *leader* connector, to discover a 'good' interaction, if it exists, and recognize it as such. Much of the reasoning in this section is inductive, driven by the inductive definition of trees. Particularly, on the *solution tree*, defined in Section 6.1.2 as a structure overlaid atop the session's connectors and components. For clarity, Figure 7.1 depicts an example of a solution tree of some session, previously shown in Section 6.1.2.

For brevity in the subsections to follow, let a sub-tree S of the solution tree be said to satisfy property P if and only if the root of S is able to recognize the set interactions which satisfy the behavioural constraints of all components in S.

1. **Session protocol constraints are compositional**
   Rather than reasoning about the session's protocol at large, the leader can instead reason whether a given interaction adheres to the behavioral constraints of all of the session's components. Essentially, we reason about the session protocol indirectly, by reasoning about the protocols of the session's constituent components directly, and relying on the compositionality of protocols for the two to coincide; see Chapter 5 for a deeper look at our notion of compositionality.

2. **Base case: components satisfy P**
   Components only occur as leaves in the solution tree. The speculation of a component eventually discovers all of the interactions which would satisfy the component's protocol, and to which state the component would advance if the interaction were realized.

   In the case of *native* components, this procedure is very simple; the speculation begins with the user application having explicitly enumerated the outcomes which they permit. Concretely, the connector stores a set of *batches* (see Section B.2.2), which enumerate

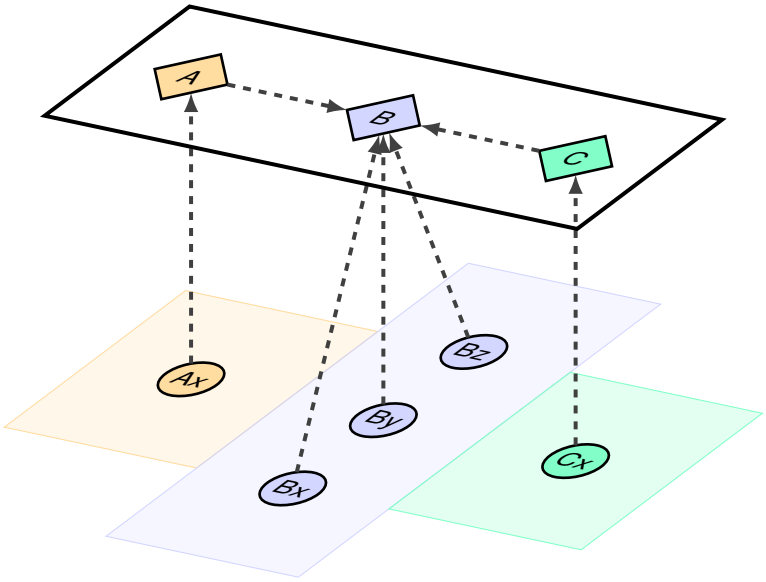


Figure 7.1: A depiction of a solution tree for a session with three connectors (diamonds) and five components (circles). Edges of the solution tree are shown as dashed arrows from child to parent; note that leaves are components, and internal nodes are connectors.

the outcomes the native permits as sets of data structures, each representing a single message exchange operation at one of its owned ports.

In the case of protocol components, its *speculative execution* explores the set of acceptable behaviors by exploring the *paths* through their protocol's control flow, as it is defined in PDL. Throughout the execution, branches are annotated by a *candidate predicate*, keeping track of the branch's requirements of the interaction whose effects are being simulated. Forks in the path are realized as forks in the components speculative execution, forking both the annotated candidate predicate, and the local variable store that encodes the component's local state. Ultimately, the set of predicates which satisfy the constraints of the component's protocol are recognized as all those satisfying the constraints of some speculative branch's candidate predicate. Note that, once forked, continued speculation of one branch has no impact on any other, as their actions are safely encapsulated to the speculative environment, and to their unique local variable store replica. For more information about speculation, see Section 6.1.2.

By case distinction, we conclude that all components satisfy P.

3. **Induction step: connectors satisfy** P
   Connectors occur at internal nodes in the solution tree. By induction, their children in the tree are roots of sub-trees, each satisfying P. For the connector to satisfy P itself, it satisfies for it to compute the *intersection* of the interaction sets known by its children. Section 6.1.2 explains how candidate predicates are the vehicle for communication of information from children to their parents in the solution tree. These structures define a procedure for computing a new predicate encoding the intersection of the interaction sets encoded by two given predicates. In a nutshell, it is driven by computing the *union* of

both constituent predicates' constraints. As new information arrives from its children, a connector is able to update a local storage of candidate predicates. In effect, connectors incrementally populate a local replica of the candidate interactions found by all nodes in the sub-tree of which they are the root.

We conclude that all connectors satisfy P.

Ultimately, the leader (the root of the solution tree) is shown to satisfy P. As the root of the entire solution tree, it recognizes the set of interactions that satisfy the behavioural constraints of all components, i.e., the behavioral constraints of the session's protocol. Eventually, if such an interaction exists, the leader will recognize it as such. By holding off the decision until this time, the runtime is certain to decide only upon an interaction that adheres to the session's protocol.

The session's state reflects the interaction

Once decided, the nature of the interaction to realize is announced to all of the session's connectors using a wave of control messages, originating at the leader, and propagated at each node to all child connectors. Concretely, the announcement encodes a candidate predicate whose constraints are sufficiently specific to uniquely identify a candidate. Section 7.1.1 explains why this candidate is also a solution, corresponding to an interaction certain to result in behavior that adheres to the session's protocol; as such, we refer to it henceforth as the *solution predicate*.

All that remains is for the state of the session (distributed over the connectors) to be updated to reflect the intended interaction. This is achieved by each connector updating the states of the components it manages to reflect the interaction's effects. For both native and protocol components, this relies upon the connector having retained the information generated during *speculation*, i.e., the connector 'remembers' which candidate predicates it previously submitted into the solution tree.

In the case of native components, the observable effects of the realization of the round's interaction are only (a) storage of the *batch index*, facilitating the application's reflection on which of their batches was successfully incorporated into the interaction, and (b) the storage of all received messages, such that the application may subsequently read the contents of messages received at their ports.

In the case of protocol components, the existence of a solution predicate guarantees that they have already computed (and retained) their local state, reflecting the effects of the corresponding interaction. Recall that the speculative execution of a protocol component explores all possible, distinct outcomes of the synchronous round, by exploring paths through the control flow of the component's protocol. The branches encoding successfully completed synchronous rounds are retained, such that their updated component state may be used to 'commit' the effects of the interaction by overwriting the component's previous (persistent) state.

We conclude that the round ends with all the session's components' states being updated to reflect the effects of the decided interaction.

7.1.2 Applications are not Starved of Control

It is a core design principle of Reowolf's notion of 'protocol' that a component's behavioral constraints be meaningful without knowledge of the component's session environment. Users and their applications rely on this property, enabling them to focus on the behavior of their native components' ports alone. By design, the behavioral constraints on the session at large may

correspond with very difficult constraint satisfaction problems, more complex than any one application may realize, and as such, may require a significant amount of time for the runtime to realize the next synchronous interaction. If left unchecked, this has the undesirable consequence of applications having to relinquish control to their connectors for an indeterminate amount of time before the round is over. Regardless of whether this truly blocks the application (or, instead, uses an asynchronous style, where the round's completion is confirmed asynchronously at a point arbitrarily far in the future), it is undesirable for applications to be able to unwittingly deadlock the session's progress without any recourse.

The connector implementation is extended, such that the application's call to facilitate the next synchronous round (`connector_sync` in the API) is parameterized with a *timeout* duration. Accordingly, the synchronization procedure is extended to take this timeout into consideration. As a result, the runtime's behavior is modified such that synchronous rounds end with one of two outcomes: (a) success, where the session's state has advanced to reflect the effects of some interaction, or (b) failure, where some application, participating in the session, has raised a timeout event before the round's interaction could be realized.

Supporting this feature leverages the control structures already in place to facilitate the synchronization procedure. Only two changes are made to the implementation:

1. **Timeout requests are forwarded to the leader**
   As part of its normal operation of their synchronization procedure, connectors block to await the progress of their network endpoints. This step is modified to bound the time spent blocked; upon un-blocking, the connector checks if the timeout duration has elapsed, since the start of the synchronous round.

   The first time (per round) a non-leader connector raises or receives a timeout request, they forward it to their parent in the solution tree.

2. **The leader decides on either failure or success**
   The first time (per round) the leader receives a timeout request prior to a decision being made, they *decide* on a failed round. As for the case of success, the result is a wave of announcement, that propagates from parent to child connector down the solution tree. The decision event per round is still unique, ensuring that the session's connectors will reach consensus on the result of the round, successful or otherwise.

This feature ensures that, while the runtime cannot guarantee that the session will always make progress, it can guarantee that an 'impatient' application will not, itself, be stuck awaiting the success of a round it would rather abandon. In the result of failure, applications have an opportunity to alter their behavior, and try something else. In the worst-case scenario, where an application is unwilling to try again, they have the ability to abandon the session, potentially forming another one in a new initial state.

The distributed nature of timeout events means that applications are subject to the impatience of the session's least-patient application. Malicious applications can inhibit progress by supplying an exceptionally short timeout duration. This is not a new problem, simply manifesting as the requirement that all components are satisfied by the interaction realized every round; a malicious application can prevent progress just the same by expressing unsatisfiable requirements. Section 12.1.3 describes future work toward removing this property.

7.1.3 The Session is Always Consistent

Section 7.1.1 shows that, per round, at most one interaction occurs, and Section 7.1.2 shows that, regardless, the synchronous round ends with all connectors reaching consensus on the

round's interaction, including the absence of an interaction (in the event the round fails, as a result of a timeout event).

The states of the session's connectors can be understood as (partial) replicas of the session's state at large, as rounds facilitate information from one connector being distributed to others. We argue that, regardless of whether the round ends successfully or otherwise, the session always has a consistent state, i.e., the connectors' local replicas of the session are all mutually consistent.

In the case of the round's failure, the safe encapsulation of the connector's speculation ensures that there are no side effects; the connector's internal state is preserved, unaffected by the work performed during the failed round. Equivalently, the connector's state 'rolls back' to that at the start of the round.

In the case of a successful round, Section 7.1.1 ensures that the announcement of the corresponding, decided interaction is unique. As a result of (a) the consideration of all components' behavioral requirements being taken into account for every component, and (b) every connector managing at least the native component, we can see that it isn't possible for two connectors' applications to be outside the synchronization procedure, such that they have observed a different number of interactions. As a result, at any time two connectors' natives are *not* participating in the synchronization procedure, their connector have realized the same sequence of interactions since the start of the session.

We conclude that regardless of whether rounds end successfully or otherwise, the session's connectors expose mutually-consistent views of the session's state to their applications.

7.2 Control Protocol Deviation

In this section, we reason about the security of the connector implementation's control algorithm. It is out of the scope of the project for this analysis to be rigorous; instead, this section provides informal reasoning, aiming to provide greater insight into the implicit relationships between a connector and its network peers, providing a deeper understanding of the nature of the implementation, and laying the groundwork for rigorous security analysis and refinement in the future work outlined in Section 12.2.13.

For brevity, we introduce some new terminology:

- **Bad connectors**
  We say a connector is bad if its participation in the session could be observed by an omnipresent observer to deviate from the connector control algorithms, described in Chapter 6; we refer to this act of deviation as *bad behavior*. Bad connectors may arise either as a result of an innocent implementation error, or as the result of malicious intent, where some process masquerades as a *good* connector in a session.

- **Trust relation**
  A component or connector *A trusts* a connector *B* if *A*'s continued observation of protocol-adherent session behavior is threatened by *B* being bad. As connectors affected by bad behavior may, in turn, act erroneously, we can understand the trust relation to be transitive. *A* trusts *B* to be good, both by acting according to the control algorithm correctly, but also not to trust any bad connectors.

7.2.1 Communication

We reason about the effects of including bad connectors in well-formed sessions, i.e, sessions which have been set up with a well-formed *solution tree*, throughout the communication phase.

A component trusts its manager

Connectors are given free reign over the states of the components they manage, and are given sole responsibility to represent their interests in the synchronization procedure. Obviously, an application written atop a bad connector cannot rely on the connector behaving as specified. This is also the case for protocol components; in general, we can understand that a bad connector has the ability to behave as though they managed any set of components at all.

As an example, a user application relying on a bad connector is subject to observing any conceivable behavior at their native components' ports, regardless of the session. This includes (a) their connector under-representing their behavioral constraints, resulting in the completion of unsatisfactory interactions, (b) the connector misrepresenting the outcome of the round, such that the user's view of the interaction is inconsistent with that of other components.

A connectors trusts its parent

During the synchronization procedure, parents in the solution tree are tasked with aggregating information from their children, and forwarding it to their parent (if they exist) or deciding accordingly (otherwise).

Leading up to the decision, bad connectors cannot be trusted to relay their children's behavioral constraints toward the leader. By abusing this trust, bad connectors may (1) under-represent their children's behavioral constraints, potentially resulting in decision of an unsatisfactory interaction, (2) over-represent their children's behavioral constraints, preventing the realization of a satisfactory interaction, potentially inhibiting the session's progress, or (3) fail to relay a child's timeout request, potentially delaying the moment an application regains control for an arbitrary duration.

Bad parents who aren't the leader may also abuse their children's trust by misrepresenting the result of the synchronous round, potentially leading to them having an inconsistent view of the session's state. This occurs whenever the bad parent forges the leader's decision announcements. For example, a bad parent can misrepresent the leader's decision, leading their children to believe that the round ended unsuccessfully when it did not, or successfully with a forged solution interaction when it did not.

7.2.2 Setup

The correct setup of the session is robust to failures to initialize channels, implemented such that failed channel creations abort the connect procedure. The only danger of bad behavior is during the latter steps of the connect procedure, described in Section 7.2.1.

1. **Tree construction**
   Correct control communications rely on connectors reaching consensus on their role in the *consensus tree* (the sub-tree of the solution tree consisting only of connectors). Ultimately, bad behavior in the formation of this control structure boil down to ways in which bad behavior can cause a partitioning in the consensus tree. This occurs when the set of

connectors fail to preserve the uniqueness of their connector identifiers, either by one connector forging others' messages, or by multiple connectors claiming identifiers determined to belong to the leader.

All other misbehavior during tree construction cannot do more harm than causing the session's setup to fail (i.e. achieving the same result as being absent). For example, a bad connector can misbehave by failing to participate in another connector's election; this behavior is essentially indistinguishable from an exceptionally slow network link, and will ultimately be treated as such by the session's components, resulting in timeout.

2. **Session transformation**
   The session transformation procedure is vulnerable in the same cases as the synchronization procedure, as they have in common that they are built atop the solution tree, with child nodes trusting their parents. In the case of session transformation, bad parents can misrepresent control messages flowing in either direction, at the expense of their children.

   The presence of a bad connector is perhaps particularly dangerous; in addition to their ability to *modify* session information as described above, they may be able to *read* session information that could be sensitive. Currently, this is not a concern, as protocols and connector states are not considered to be sensitive information. If this changes in future work, care should be taken to minimize the information a (good) connector provides to its children. For example, once the leader performs the session transformation on the `SessionInfo` map structure (see Section 8.1), parents can *partition* session information such that children receive only information relevant to the connectors in their sub-tree of the consensus tree, rather than parents simply broadcasting the entire session's information indiscriminately.

7.2.3 Malformed or Inappropriate Messages

Generally, the connector runtime is robust to the receipt of control messages that are malformed, or arrive when they are in an inappropriate state, or from an unexpected neighbor. For this reason, it is generally not dangerous for the session to include bad connectors unless they are trusted by good connectors, in the manner described the previously.

In cases where a good connector receives malformed or in appropriate messages that can be explained as a result of delays in the network, such messages are silently discarded without any harm (e.g. speculative messages from round N arriving once the connector has entered round $N + 1$). Otherwise, the implementation relies on the ordering and integrity guarantees of the TCP transports to conclude that the sender is a bad component, raises an error message, and terminates the session. For example, in the event a connector receives a leader's decision announcement during the setup phase, they can conclude that some connector is behaving erroneously; a decision should be impossible without their prior participation in the round, which is impossible while the connector is in the setup phase.

Chapter 8

Session Optimization

This section goes into greater detail about the properties of the connector implementation that aim to improve its performance characteristics, particularly during the communication phase. The effectiveness of these optimizations is tested in Chapter 11.

Section 8.1 describes the *session optimization* procedure, in which the set of connectors transform their internal representation of the session in its initial state, such that its work in the communication phase is more efficiently performed. Section 8.2 describes particularities of the ways connector manage the message payloads passed between components to facilitate cheap in-memory movement and replication.

8.1 Session Transformation Procedure

The performance of an application using connectors for communications depends on the particulars of the work performed during the *synchronization procedure* (see Section 6.1), which realizes the user's communications. Until the results of this procedure are finalized, connectors generate and aggregate speculative data, and great care is taken to avoid these tasks 'leaking' to the user. Consequently, the user program cannot observe the majority of this speculative work, with the exception being the resources utilized to complete it, such as wall time, and system memory. The observation is that many sessions with different internals and performance characteristics ultimately expose the same *observable behavior* to the user. We define *observable behavior* as the messages the user's application sends and receives through its interface ports, as this is its only defined means of participating in the session's interactions.

This section describes the *session transformation* procedure, in which the set of a session's connectors transform their initial configuration without affecting its observable behavior. Concretely, this is realized by the addition of an extra, final step to the *connect procedure* (see Section 6.2.3), involving the exchange, aggregation, transformation and redistribution of configuration data between the session's connectors. Generally, we are interested in using session transformations to optimize the session's runtime characteristics, such as reducing the time necessary to complete the synchronization procedure, or reducing overall network traffic.

Figure 8.1 gives a simple example of two sessions with different initial configurations, but whose *x* and *y* components observe identical behavior. Presuming the latter session can communicate more efficiently than the former, (on account of having a simpler configuration, requiring fewer resources), an effective session transformation would enable sessions provided the complex configuration to have the performance characteristics of the simpler one.

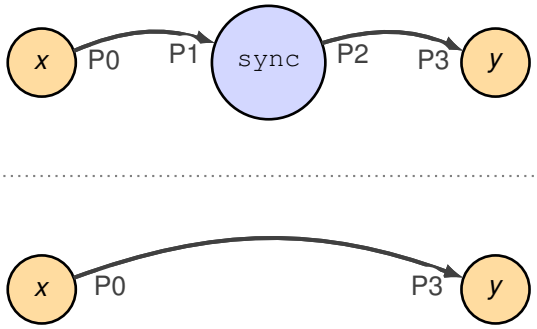


Figure 8.1: Component graph of two sessions (above and below). Despite the sessions differing in the number and arrangement of ports and components, they have identical observable behavior with respect to *x* and *y*.

The session optimization takes advantage of the *consensus tree* overlay network (see Section 6.1.2) already available. In a nutshell, the procedure revolves around connectors exchanging information about their local, initial configuration via a (de) serializable structure: `SessionInfo`. The distributed procedure emerges from each connector process participating the following sequence of steps:

1. **Gather**
   Each connector process receives a session information message from each of its children in the overlay network, aggregates it, and sends the result to its parent if they exist.

2. **Transform**
   The leader, characterized by the lack of a parent in the overlay network, interprets the session information message as a snapshot of the entire system. It traverses and modifies the contents of the snapshot. This step is responsible for identifying opportunities for optimizations, and applying them to the session.

3. **Scatter**
   Until it has the modified session information, the connector process awaits its arrival from their parent, whereupon it forwards[1] the information to each of its children.

4. **Apply Transformations**
   Each connector process interprets the modified session information, and applies modifications to their internals as appropriate. For fully serializable structures (e.g. persistent protocol data seen in Section 6.2.1), this can be as simple as overwriting the connector's existing structure. Other structures require that the session information communicate modifications to be applied as symbolic instructions (E.g. the session information may instruct a particular connector process to create a set of new transport-layer endpoints given the relevant setup information such as a remote IP address).

While all other steps facilitate the centralization of this distributed procedure, the work of the leader in the aforementioned 'transform' step realizes the session transformation. While a robust and generically-applicable transformation procedure is left for future work, described

[1] The scattering of session information takes the form of a broadcast in this version of the implementation, effectively giving all connector processes a snapshot of the entire system's state at the end of the setup phase. If desired, this procedure can be modified to narrow down the contents of the scatter messages, such that a subset of the entire system's information reaches each connector process (as long as it suffices for them to transform their local internals).

fully in Section 12.2.1, we provide a proof-of-concept for the power of the session transformation procedure in the form of several examples and corresponding benchmark in Section 11.2.

8.2 Local Optimization

This section details the implementation and workings of *local optimizations*, not realized explicitly through the session optimization procedure described in Section 8.1. Rather, these optimizations are a result of the systemic properties of connectors encapsulating the behavior that would otherwise be spread over several transport endpoints, for example, in the case of TCP sockets. These optimizations are concerned with mitigating the costs of movement and replication of large messages, as this is the only task that connectors and sockets have in common, motivating the use of sockets as a point of reference.

Section 11.1 shows examples, and performs benchmarking experiments to demonstrate the effectiveness of these local optimizations.

8.2.1 Lightweight Local Message Passing

To span the network between distributed components, there is no getting around the use of the OSI transport layer for logical message exchange. However, where the notions of 'transport channel endpoint' and 'logical channel endpoint' coincide in BSD-style sockets, this is not so for connectors. Applications and protocol components alike are able to create logical channels without the added costs associated with network messaging. Applications in particular can achieve this either implicitly (by creating both ends of a channel separately with `connector_add_net_port`), or, more conveniently, explicitly with `connector_add_port_pair`. Messages exchanged between components managed by the same connector are done so *symbolically*, i.e., by reference, with the contents of the message kept in-place, exploiting the benefits of shared memory whenever possible. Concretely, the byte buffer storing a message's contents is allocated on the heap, and its movement through the connector's internals are achieved by moving a reference, with the heap allocation left in-place.

8.2.2 Lightweight Local Message Replication

Many component networks with interesting behavior express their work in terms of very many message passing operations. If messages are large, a naïve implementation might require very many `memcpy` operations to replicate and move messages between components safely. Generally, it is unsafe for logically-distinct messages to be represented as aliases to the same message contents without introducing the kind of subtle and deadly bugs that result from circumstantial race conditions.

The implementation makes use of a memory- and thread-safe mechanism for reference-counted message passing. Building on the functionality described in Section 8.2.1 above, the message structures passed between components are small, symbolic representations to *atomic reference counted* message contents kept in-place. Replicas of the same message contents can be created and discarded cheaply, *aliasing* the contents without introducing data races. Components can pass around these replicas repeatedly, only duplicating the message contents when the contents of the replicas diverge, i.e., using a *copy-on-write* pattern.

Currently, serialization and transmission of these aliased messages is a barrier to this optimization, but Section 12.2.2 discusses a generalization without this limitation.

Chapter 9

Backwards Compatibility

This chapter explores extensions to the reference implementation to facilitate ergonomic interoperability with applications built atop BSD-style network sockets. Although many forms of sockets exist, some of which do not even concern network programming, this work focuses primarily on UDP. For now, this serves as a practical decision; implementation effort is minimized by focusing on only a subset of the available kinds of socket. Furthermore, in works to follow, the choice of UDP translates more directly to an implementation built atop the network layer, sending and receiving IP packets in kernel mode.

In the grand scheme of things, the goal of Reowolf is to replace BSD-style sockets with connectors for message passing over the internet. This is no small task, as sockets have been deeply entrenched into existing software, and into the minds of programmers for decades. As such, our approach is to support backwards-compatibility with sockets, such that connectors may be adopted into new and existing codebases with minimal friction. This chapter describes an impmentation of an idea referred to as the 'socket sandwich', where connector sessions are built atop sockets in the transport layer below, while also exposing an API which mimics that of sockets to the application layer above. The goal of future work is for these *pseudo-socket* connectors to detect fellow connectors in the network, regardless of their users' API, and to compose into distributed connector sessions.

The complementary halves of the socket sandwich are described in complementary sections. Section 9.1 describes *UDP mediator components*, components employed to mediate communications over a UDP channel with some remote peer. Section 9.2 describes the pseudo-socket API, which allows applications to use connectors as they would use conventional UDP sockets for network programming.

The usages of these features is demonstrated in Section 10.3 to follow, concretely, giving examples of C programs making use of these various APIs for network communications with one another.

9.1 UDP Mediator Components

This section explains UDP mediator components, the 'connectors atop sockets' half of the socket sandwich, allowing connectors to communicate with networked peers via UDP channels. To avoid confusion, this section distinguishes the terminology used when referring to different notions of 'message':

- **Datagrams** are variable-length message sent over a UDP channel in the transport layer, corresponding closely (but not identically) to packets over IP in the network layer.
- **User messages** are variable-length byte sequences, send and received over the network. Users of both UDP sockets and connector components reason about the exchange of these messages over the network.
- **Control messages** are messages used internally by the by the distributed connector runtime to drive various distributed algorithms that facilitate synchronization, consensus and so on, described in Chapter 6. These messages may facilitate the exchange of user messages, but they need not correspond 1-to-1.

  This relationship is similar to that between the (user-facing) byte stream of TCP, which is facilitated by the exchange packets over IP annotated with control information such as sequence numbers and checksums.
- **Speculative messages** are the kind of control message which Section 6.1 explains as being the connector runtime's vehicle for exchanging speculative port information. After consensus is reached, it is determined whether or not each speculative message encodes a user message.

Naturally, all these forms of message are related. In the case of programs written atop sockets, each datagram's message payload encodes a user message. A program written atop connectors might exchange a sequence of user messages, which the connector runtime realizes as a set of control messages, some of which are also speculative messages.

9.1.1 User View

The connector API is extended to include `new_udp_mediator_component`, which allows for the initialization of a *UDP mediator component* given a pair of socket addresses to characterize both endpoints of a UDP channel. During the communication phase, such components participate in synchronous message exchange through their ports in the usual fashion. The behavior of the component itself is that all messages passing through are simply forwarded either into or from its UDP socket. In this fashion, UDP mediator components may be used to communicate with the world beyond the connector. An example of a session topology including a UDP mediator component is depicted in Figure 9.1.

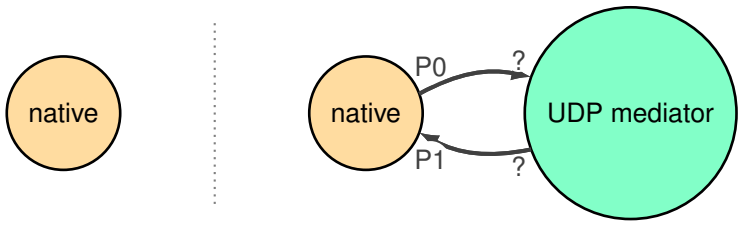


Figure 9.1: Component graph of a session with a native component able to communicate accross the connectors' session boundary via ports to the UDP mediator component, which manages a UDP endpoint.

9.1.2 Connector View

UDP mediator components have in common with native components that they represent the interface between the distributed connector runtime and the outside world. Realizing the behavior of a UDP mediator component requires the connector runtime to perform message translation tasks in two directions: translating incoming datagrams / data messages into control messages, and vice versa.

Translating control messages to datagrams

Recall from Chapter 6 the connector runtime works to shield native components from its inner speculation procedures. Control messages representing speculative data messages are buffered until the round's completion, whereafter the system has reached consensus on the oracle that determines the new consistent distributed state, including the data messages that were synchronously exchanged at ports. Protecting the component is relatively straightforward, as all the necessary information is retained in the control message; speculative messages are buffered until the moment of consensus, whereafter the set of user messages is determined.

This is the case for UDP mediator components also. They differ from native components only in what happens to this set of user messages. For native components, they are accessible for reading via the connector API's `read_gotten` procedure. For UDP mediator components, these user messages are wrapped into datagrams, and sent over the network.

Translating datagrams to control messages

Where translation of control messages to datagrams is a lossy procedure (as it represents the collapse of control information), translation of datagrams to control messages necessitates the injection of absent control information. Fortunately, UDP has very few reliability guarantees to preserve, such that many distinct implementations are all equally correct. For example, as UDP provides no guarantees on the ordering or arrival time of datagrams, it suffices for connectors to associate incoming datagrams with any round number no smaller that of the round in which they are received. For simplicity, the reference implementation uses the number of the round in which the datagram is received.

The reference implementation has a limitation that prevents UDP mediator components from translating UDP traffic in the most straightforward manner. Namely, ports can transport at most one user message per synchronous round. Chapter 12 describes future work that aims to remove this limitation. For now, the reference implementation abuses UDP's lack of delivery guarantees by discarding all but the first datagram received.

An interesting alternative is an implementation that buffers all datagrams received during the synchronous round, and for each, creates a mututally-exclusive speculative message; the result of this approach is a UDP mediator component which appears to 'guess' satisfactory incoming messages, with any others silently discarded, as if they never arrived at all. This implementation is correct also, as the resulting datagram loss is permissible UDP behavior.

Another interesting implementation is achieved with a minor adjustment on the former: every round, the UDP mediator component creates a set of mutually-exclusive speculative messages, one for every datagram received during the session. The resulting behavior is a UDP mediator that replays whichever data message is opportune for the round reaching consensus, at the cost of introducing message duplication above and beyond that already introduced by the UDP channel itself.

9.2 Pseudo-socket API for Connectors

This section details the *pseudo-socket* API connectors expose to application programmers, providing the 'sockets atop connectors' half of the socket sandwich. This API aims to mimic that of conventional BSD-style UDP sockets, such that application developers may swap out their sockets for connectors with minimal modification of their program logic. For now, this serves to demonstrate the feasibility of replacing existing socket programming logic with that of connectors. More importantly, this lays the groundwork for connectors being employed transparently, where Reowolf-enabled peers instantiate connectors in place of sockets, gaining access to the features of connectors on demand. Until these optimizations are in place, the effects of using the pseudo-socket API are straightforward: every created socket creates the connector shown in Figure 9.1.

9.2.1 Behavioral Differences Between Connectors and UDP Sockets

Building socket behavior from connectors necessitates reconciling the differences between the ways they behave, and the ways their functionality is represented in their API. In this section, these differences are examined in a structured manner, such that they may serve as reference for the sections to follow:

1. **UDP sockets are not strictly connection-oriented**
   The UDP transport protocol is essentially stateless, oriented around the sending and receipt of datagrams between endpoints. Sockets used in this way might make frequent use of procedures such as `sendto`, where one specifies the recipient peer's socket address on a case-by-case basis.

   To address a frequent use case of UDP, the socket API facilitates connection-oriented usage through procedures `connect`, `send` and `recv`. The former specifies the peer address of all future communications, such that messages can be sent and received without a given or returned peer address.

   It is important to note that the socket allows users the ability to make partial use of the connection-oriented paradigm; UDP sockets can be connected at any time, and can be reconnected repeatedly to overwrite the peer address for subsequent sends and receives. This is not the case for connectors, for which `put` and `get` are always sent over communication channels whose connection persists implicitly. Components do not have the authority to arbitrarily change the network address or identity of their ports' peers.

2. **Users identify sockets with file descriptions**
   The reference implementation stores and manages its connector structures in user space, relying on the operating system to drive the exchange of control messages through transport layer endpoints. Consequently, the connector API accesses connectors via pointer.

   In contrast, the bulk of sockets' functionality is provided by the operating system, which manages and stores socket structures itself, and allows users in the application layer access via *file descriptors*, integer identifiers used as keys to the appropriate socket data structure in a storage managed by the operating system.

3. **Socket operations are completed by the operating system**
   As sockets are implemented primarily in the operating system kernel, the user's thread plays only a small part in managing the resources of the socket. This is not obvious while sockets are used in blocking mode, where the loss of control flow may as well be a result

of the user thread performing the work itself. By contrast, connectors are implemented in user space, and thus the caller thread does indeed perform a significant portion of managing the state of the connector (safely encapsulated within the `sync` call). This difference between the way sockets and connectors are implemented becomes clear when sockets are used in non-blocking mode, which allows the user to regain control of their thread while message exchange operations are completed by the operating system.

4. **The socket API is thread safe at runtime**
   The connector API is specialized to the task of synchronized network communications, using structures in user space. As such, mutually exclusive access per connector is achieved trivially, requiring conscious effort to circumvent with users employing multi-threading. As such, the connector API relies on users to employ their own mutual exclusion solution for these exceptional cases.

   In contrast, sockets are protected by locks to make resource access via file descriptor threadsafe in general, which includes access of resources available to all host processes, such as files in the file system.

5. **The unit of synchronization and the message channel coincide in sockets**
   For connectors, there is a meaningful grouping of isolated *actions* into *interactions*; we say the actions within each interaction are *synchronous*. This is reflected in the connector API in the distinction between actions at the granularity of ports, (e.g. `put`), and interactions at the granularity of the connector (e.g., `sync`). In contrast,we say that sockets are *asynchronous*, because every message exchange operation is part of its own singleton set of synchronized actions. For this reason, procedures in the socket API require the user only to identify the message channel, with which the unit of synchronization coincides.

   Note that our usage of the term ‘synchronous’ in this context should not be confused with another usage relevant in the context of network programming. To clarify, blocking socket operations may also be considered synchronous, in that the user’s `send` or `recv` procedure is synchronized with the operating system’s completion of the message exchange.

9.2.2 Limitations of the Pseudo-Socket API

This section addresses conscious choices to limit the pseudo-socket API, minimizing its complexity, and ensuring that its implementation is sound and in service of the goals of the milestone.

Connection-less message operations are not supported

Recall from the previous section that the socket API allows for a mix of connection-oriented and connection-less usage. The pseudo-socket API supports only a connection-oriented usage of UDP sockets, such that their behavior is more in line with the behavior of connectors’ ports: an opaque endpoint able to send or receive payloads over a message channel.

Facilitating a connection-less API efficiently and accurately would require the development of features currently planned for the follow-up project to Reowolf, described in Section 12.1.1. For example, a pseudo-socket re-connecting to a new peer between message exchanges may be realized as the creation of a new network port, with a new peer. This ability to extend the participants of the session introduce problems for the reference implementation, which is built around a session-wide logical clock, for which the addition of new participants is not supported.

Message operations are blocking

UDP sockets in non-blocking mode have in common with connectors that initiation of message exchange operations does not block; i.e., neither `send` with a non-blocking socket, nor `put` with a connector's port will ever block the user thread. They also have in common that their respective APIs offer a blocking call for awaiting the completion of message exchanges, polling with some selector for the former, and `sync` for the latter. For the sake of simplicity, henceforth we refer specifically to the `epoll` selector system, available to platforms Linux, Solaris and Android, and offering `epoll_wait` as the canonical means of blocking the caller thread until message operations have been completed.

The pseudo-socket API cannot trivially mimic `rw_epoll_wait`, the pseudo-socket equivalent to `epoll_wait` out of invocations to `sync`. The problem is the result of two factors: (a) `sync` drives the message exchanges of a single connector's ports, while `rw_epoll_wait` accepts an arbitrary set of pseudo-sockets whose message operations complete in parallel, and (b) `sync` is blocking, as the caller's thread itself drives the progress of message exchanges.

As is often the case, there are multiple approaches to building non-blocking behavior from blocking primitives, each approximating the ideal result differently; here are some examples:

1. **Subdivide the timeout duration over `sync` calls**
   The user's thread can invoke `sync` for the relevant set of connectors in turn, returning an event once a call successfully completes a synchronous round. The work of synchronizing connectors can be multiplexed over the available timeout duration. This approach runs into the problem of imposing an ordering on the connectors, such that an expensive synchronization may impede the success of successful synchronization of those after it. Users may perceive `rw_epoll_wait` as slow and unreliable.

2. **Call `sync` in parallel using worker threads**
   The prior example's problem of introducing an undesirable ordering can be circumvented by creating a pool of worker threads such that each connector is sychronized by a dedicated worker in parallel. The problem with this approach is that the entire procedure must wait for its slowest worker; where `epoll_wait` returns early if message operations are completed, just one slow worker's thread would be stuck in their `sync` call, possibly until the end of the timeout duration. As with the previous approach, the resulting implementation is slow, particularly in cases where users expect very little overhead, for which the work of creating or managing a threadpool will dominate the work time.

3. **Circumvent `sync`**
   As the blocking nature of `sync` is the common denominator for problems with supporting non-blocking polling, an effective approach forgoes the usual connector synchronization procedure for one designed with non-blocking polling in mind. However, this approach fails to provide meaningful interoperability between connectors and sockets in the spirit of the milestone. Taken to the extreme, this customized implementation might as well access the UDP sockets contained within the connectors' UDP mediator components and poll them to a selector with `epoll_wait`.

Ultimately, non-blocking pseudo-sockets are best realized with a future connector API able to drive session communications without blocking the caller thread. Most likely, this will be achieved in a manner similar to that of sockets today, where the completion of message exchange operations is performed by the operating system. This development is scheduled for future work in the follow-up to the Reowolf project, and is described further in Section 12.2.8.

9.2.3 Pseudo-Socket Implementation

This section details the pseudo-socket implementation in detail, explaining how it can be used, and how it builds the behavior of UDP sockets from that of connectors.

Data structure and thread safety

For historical reasons, file descriptors are often smaller than machine words, prohibiting a scheme that trivially represents connectors with file descriptors by 'stuffing' each with a connector pointer.

The implementation of the pseudo-socket API mimics the role of the operating system in conventional sockets, introducing `CC_MAP` as a manager of file descriptor allocations, and a mapping from allocated file descriptors to `connector complex`, a connector bundled alongside some metadata. `CC_MAP` is protected by a coarse-grained readers-writer lock, for which mutual exclusion is only necessary in order to allocate and free mappings and file descriptors. Mutual exclusion per connector complex is achieved with another fine-grained lock. Consequently, pseudo-socket operations are thread-safe, and operations on different pseudo-sockets can occur concurrently.

As this storage is subject to change in the follow-up to the Reowolf project, it was decided to focus on simplicity and correctness in favour of performance optimization. Section 12.2.7 describes future work toward 'pushing' the connector implementation into the operating system's kernel, which would result in a re-design of the storage structure for connectors.

Avoiding namespace conflicts

The pseudo-socket API defines a set of procedures which mirror those useable for UDP sockets, primarily[1] defined in `sys/socket.h`, a C header file considered canonical for systems-level network programming. Recall that connectors themselves are currently built atop sockets. As such, it is not possible to replace the definitions of essential socket operations such as `socket` and `send` without breaking the connector implementation.

Our approach populates the namespace with identifiers for socket procedures and their pseudo-socket counterparts, where those of the latter are distinguished from those of the former only by the addition of a `rw_` prefix ('rw' for 'Reowolf'). As a result, the connector implementation continues to work using conventional sockets, but users are able to switch from the socket and pseudo-socket API by use of a simple string replacement macro which prepends `rw_` to the identifiers of socket procedure calls.

Pseudo-socket procedures

As is the case for connectors, pseudo-sockets begin in the setup phase, and transition to the communication phase once their configuration is complete. In addition to a small set of constants for the purposes of error-handling, the pseudo-socket API is comprised of a set of procedures, with identifiers, declarations and behaviors mirroring those of the socket API:

```
#include <sys/socket.h> // defines {sockaddr, socklen_t}
int rw_socket(int domain, int type, int protocol);
```

[1] The `close` procedure is defined in `unistd.h`, as it is used for closing file descriptors in general, and not just those representing network sockets.

```
int rw_connect(int fd, const struct sockaddr *address, socklen_t
↪  address_len);
int rw_bind(int socket, const struct sockaddr *address, socklen_t
↪  address_len);
int rw_close(int fd);
ssize_t rw_send(int fd, const void * message, size_t length, int
↪  flags);
ssize_t rw_recv(int fd, void * buffer, size_t length, int flags);
```

Conveniently, all of the implemented procedures can be partitioned, such that each part consists of a reciprocal pair of procedures.

The first part allows pseudo-sockets to be created and destroyed:

- **`rw_socket`**
  A writer lock is acquired on `CC_MAP`, allocating a new file descriptor key and initializing a new trivial connector, which remains in the setup phase until it is both connected and bound to socket addresses. If the given file descriptor is already in use for a connector complex, an error is returned.

- **`rw_close`**
  A writer lock is acquired on `CC_MAP`. If a mapping is associated with the given file descriptor, the associated connector complex is destroyed and its resources freed. If the connector is in the connected phase, its UDP mediator socket is destroyed, and its UDP socket is closed. If no mapping is associated with the file descriptor, an error is returned.

The second part consists of procedures for configured pseudo-sockets in their setup phase:

- **`rw_bind`**
  A reader lock is acquired on `CC_MAP`. If the given file descriptor is unmapped, an error is returned. Otherwise, the mapped connector complex is locked. If the connector is not yet in the communication phase, the given local socket address is stored, otherwise an error is returned. If both local and peer addresses are available, the connector transitions to the communication phase, bound to a single UDP mediator component in the configuration depicted in Figure 9.1. The putter and getter ports linking the connector's native component to the UDP mediator are stored in the connector complex.

- **`rw_connect`**
  This procedure is identical to that of `rw_bind`, except for storing the given socket address as the peer address. Consequently, these two procedures can be called in any order, and the connector will complete its setup once both addresses are provided.

The third part exposes message exchange procedures, only available to pseudo-sockets in the communication phase:

- **`rw_send`**
  A reader lock is acquired on `CC_MAP`. If the given file descriptor is unmapped, an error is returned. Otherwise, the mapped connector complex is locked. If the connector is still in the setup phase, an error is returned. Otherwise, the connector attempts to complete a single synchronous round, with the native component providing only the option of putting the user-supplied message into the stored putter port. As the connector's session is known to correspond with that depicted in Figure 9.1, the synchronous round always succeeds, with the result connector's single UDP mediator component emitting the user's message into the network in the form of a datagram.

- **`rw_recv`**
  As its dual, `rw_recv` is implemented similarly to `rw_send`, differing only in that rather that including a `put` using the stored putter port, it includes a get with the stored getter port. The `sync` call is given no timeout, and the only permitted solution requires the native component to receive a message. The effect is `rw_recv` blocking until it succeeds, whereafter the contents of the message are accessed in the usual fashion, and copied into the user's supplied message buffer; as is the case for `recv`, in the event the buffer is too small to contain the entirety of the message, it is truncated to fit.

Chapter 10

Examples and Usage

This chapter provides an 'external' view of the contributions of the Reowolf project. This includes a bottom-up approach to writing application and protocol code for building sessions, given in Section 10.1. Section 10.2 expands on protocol programming in PDL, and showing its correspondence with PDL's predecessor language: Reo. Finally, Section 10.3 shows how connectors can mimic sockets, enabling application programmers to incrementally port their applications written in terms of connection-oriented UDP sockets to connectors piece by piece.

10.1 Programming with Connectors

This section gives application developers an entry point into network programming with connectors and PDL, starting with simple use cases of local and two-party communications, to complex cases, which make use of synchronous, multi-party communications, and protocol components.

10.1.1 Interaction-Based Message Passing

The most fundamental primitive for communication of messages is provided by *sync*, the synchronous channel, which transmits a message from a source to a destination in a manner similar to that of IP or UDP. Omitting only the necessary import directives, the following example in the C programming language demonstrates an application creating a connector with a single synchronous channel, exposed as two ports, one for each channel endpoint.

```
void main() {
    // create, configure, connect
    Arc_ProtocolDescription * pd = protocol_description_parse("", 0);
    Connector * c = connector_new(pd);
    PortId x, y;
    connector_add_port_pair(c, &x, &y);
    connector_connect(c, -1);
}
```

Two distinct processes can cooperate in the creation of a shared session by cooperating in the creation of at least one synchronous channel by creating and connecting its endpoints separately. The following example shows two procedures, one for each process, sharing a session in which they are spanned by a single synchronous channel.

```
void amy() {
    Arc_ProtocolDescription * pd = protocol_description_parse("", 0);
    Connector * c = connector_new(pd);
    PortId port;
    connector_add_net_port(c, &port,
        // network binding info...
        (FfiSocketAddr) {{127, 0, 0, 1}, 8000},
        Polarity_Putter, EndpointPolarity_Passive);
    connector_connect(c, -1); // -1 for infinite timeout
}
```

```
void bob() {
    Arc_ProtocolDescription * pd = protocol_description_parse("", 0);
    Connector * c = connector_new(pd);
    PortId port;
    connector_add_net_port(c, &port,
        // network binding info...
       (FfiSocketAddr) {{127, 0, 0, 1}, 8000},
        Polarity_Getter, EndpointPolarity_Active);
    connector_connect(c, -1); // -1 for infinite timeout
}
```

Unlike IP, communication occurs between participants in an explicitly constructed session, such as that used by TCP. The session is maintained by the runtime, and all endpoints and channels are localized to the session, and maintained until it is shut down. During the session, applications may have their connector structure prepare and complete message exchange actions to facilitate message passing.

```
void amy() {
    /* setup of connector `c` and port `port` omitted */
    connector_put_bytes(c, port, "hello", 5);
    connector_sync(c, -1); // -1 for infinite timeout
}
```

```
void bob() {
    /* setup of connector `c` and port `port` omitted */
    connector_get(c, port);
    connector_sync(c, -1); // -1 for infinite timeout
    // connector_gotten_bytes(c, port, 0) returns "hello"
}
```

The runtime shares with Reo its focus on interaction-based communication; all message exchange actions succeed only if they are part of a well-formed distributed interaction. Most essentially, this guarantees that messages are sent if and only if they are also received. For maximal flexibility, applications may contribute actions on-the-fly, and rely on the runtime's best effort to identify and realize well-formed interactions just in time. For a simple example, an application may offer a message to an unknown peer, and reflect on whether the action succeeded afterward.

```
void amy() {
    /* setup of connector `c` and port `port` omitted */
    connector_put_bytes(c, port, "hello", 5);
    int err = connector_sync(c, -1);
    // `err` returns a negative integer (indicating an error).
    // No interaction succeeded!
}
```

```
void bob() {
    /* setup of connector `c` and port `port` omitted */
    // bob does NOT call connector_get(...);
    int err = connector_sync(c, -1);
    // `err` returns a negative integer (indicating an error).
    // No interaction succeeded!
}
```

Even without making significant use of components defined in PDL, applications may be configured with any number of ports, enabling message exchange through any number of distinct message channels. In this fashion, any number of hosts can participate in multi-party communication sessions.

```
void main() {
    // create, configure, connect
    Arc_ProtocolDescription * pd = protocol_description_parse("", 0);
    Connector * c = connector_new(pd);
    PortId p0, p1;
    connector_add_net_port(c, &p0,
        (FfiSocketAddr) {{127, 0, 0, 1}, 8000},
        Polarity_Getter, EndpointPolarity_Passive);
    connector_add_net_port(c, &p1,
        (FfiSocketAddr) {{127, 0, 0, 1}, 8001},
        Polarity_Getter, EndpointPolarity_Active);
    connector_connect(c, -1);

    // Get a message through `p0`
    connector_get(c, p0);
    connector_sync(c, -1);

    // Get a message through `p1`
    connector_get(c, p1);
    connector_sync(c, -1);
}
```

Sets of messages may be explicitly synchronized, ensuring that either they all succeed as part of the same interaction, or not at all, allowing applications to participate in interactions that involve any number of participants. The following application synchronizes the sending of two messages through two distinct ports, such that they succeed or fail together.

```
void main() {
    /* setup of connector `c` and ports {`p0`, `p1`} */
    connector_put(c, p0, "former", 6);
    connector_put(c, p1, "latter", 6);
    connector_sync(c, -1);
}
```

Applications can express non-deterministic choice by delimiting the boundaries between ‘batches’ of message operations, expressing that the operations of exactly one batch should succeed. Offering more options facilitates a larger set of potential interactions to the connector runtime, potentially enabling the system to progress even when some actions cannot be arranged into an interaction. For a simple example, consider how an application offering an optional message facilitates the system to progress regardless of whether their peer accepts the message. The application can reflect on the return value of `connector_sync` to determine (by index) which of their batches succeeded, and thus, whether their message was exchanged.

```
void amy() {
    /* setup of connector `c` and port `port` omitted */
    connector_next_batch(c); // ZERO actions in batch 0

    connector_put_bytes(c, port, "hello", 5);
    int err = connector_sync(c, -1); // ONE actions in batch 1
    // if err==0, no message was sent
    // if err==1, "hello" was sent
}
```

The combination of synchrony and non-determinism is sufficient for the solution of various high-level problems oriented around constraint satisfaction. In a sense, applications can employ the connector runtime as a constraint solver by encoding the satisfaction of their problem as the well-formedness of a communication interaction. For example, consider the all-pairs matching problem: a graph of N vertices connected by an arbitrary is satisfied by a solution, a subgraph containing all vertices such that all vertices have one incident edge; this sort of problem pairs the set of vertices with edges, considering only the edges given, which can be applied to matchmaking of any kind, including online two-player games, dating websites and so forth. The following C program shows one component's expression of their local constraints; the result of a synchronous round encodes the solution to the corresponding distributed constraint satisfaction problem:

```
void main() {
    PortId ports[3];
    Connector c;
    /* setup omitted */
    connector_get(c, port[0]);
    connector_next_batch(c);

    connector_get(c, port[1]);
    connector_next_batch(c);

    connector_get(c, port[2]);
    int err = connector_sync(c, -1);
    // err result in {0,1,2} encodes the identity of my peer
}
```

10.1.2 Making Explicit the Protocol

The previous examples demonstrate the essence using connectors to facilitate interaction-based communication over the internet. An essential tenet of Reo-style coordination is exploiting the constraints on achievable communication made possible by having access to an explicit *protocol*. The connector runtime enables the expression of such a protocol piecemeal,

as the creation of protocol components, which participate in the communication through ports in a manner defined by their given protocol, expressed in PDL. This approach to concurrent programming is similar to the actor model; in both cases, units of work are expressed as the tasks of actors, related through messages they exchange. During either setup or communication phases, the application may instantiate new protocol components defined in their configured protocol. These new components accept a set of ports moved from the interface of the application's component; in this way, new protocol components are 'threaded' into the session.

In the following example, the application threads a component instantiating the user-defined `force` protocol, which specifies the component's behavior such that it forwards a message from its output to its input each round. By including this protocol component, the set of possible interactions permitted in this session at runtime is constrained to just those in which a message flows through the channel.

```
char g_pdl[] =
"primitive force(in a, out b) {                    "
"    while(true) synchronous { put(b, get(a)); }"
"}                                                 ";
void main() {
    Arc_ProtocolDescription * pd =
        protocol_description_parse(g_pdl, sizeof(g_pdl));
    Connector * c = connector_new(pd);
    PortId p0, p1, p2;
    connector_add_port_pair(&p0, &p1);
    connector_add_net_port(c, &p2,
       (FfiSocketAddr) {{127, 0, 0, 1}, 8000},
        Polarity_Putter, EndpointPolarity_Active);
    /*  [main]
        p0  p1  p2---->(network)
        |   ^
        `---`                              */
    connector_add_component(c, "force", (PortId[]){p1, p2}, 2);
    /*  [main]
        p0
        |
        `-->p1 [force] p2----> (network) */
    connector_connect(c, -1);
}
```

Owing to its high-level nature, work expressed as PDL components is expressed at a level suitable to the task of distributed, interaction-based coordination, such that tasks may be expressed more intuitively. Consider the example of a three-party system, in which one host generates messages always replicated to the remaining two peers. At runtime, the connector runtime will guarantee that all components participate protocol-adherent behavior.

In the following example, the application instantiates a `replicator2` component as an intermediary between `p0`, the only port retained for the application's use, and ports `p2` and `p3`, which traverse the network to some anoymous peer(s). The topology of the channels and the definition of the protocol guarantees that the remote peers will certainly always receive identical

messages, regardless of the implementation of the application itself. Note that `replicator2` is not visible in the C source, as it is defined by default in Reowolf's standard library.

```
primitive replicator2(in a, out b, out c) {
    while(true) synchronous {
        if(fires(a)) {
            msg m = get(a);
            put(b, m);
            put(c, m);
        }
    }
}
```

```
void main() {
    Arc_ProtocolDescription * pd = protocol_description_parse("", 0);
    Connector * c = connector_new(pd);
    PortId p0, p1, p2, p3;
    connector_add_port_pair(&p0, &p1);
    connector_add_net_port(c, &p2,
       (FfiSocketAddr) {{127, 0, 0, 1}, 8000},
        Polarity_Putter, EndpointPolarity_Active);
    connector_add_net_port(c, &p3,
       (FfiSocketAddr) {{127, 0, 0, 1}, 8001},
        Polarity_Putter, EndpointPolarity_Active);
    /*  [main]
        p0  p1  p2  p3
        |   ^    |   |
        `---`    V   V
                (network)        */
    connector_add_component(c, "replicator2",
        (PortId[]){p1, p2, p3}, 3);
    /*  [main]    [replicator2]
        p0-------->p1  p2  p3
                       |   |
                       V   V
                      (network)  */
    connector_connect(c, -1);
}
```

In cases of more complex protocols, the usage of components instantiated with a predetermined protocol description may represent a significant reduction in implementation effort on the part of the application developer. The delegation of coordination logic to the definition of the protocol encourages a sanitary programming style, in which coordination logic is extracted out of user applications, and relegated to the protocol definition, making clearer the separation of concerns between distributed coordination, and local computation. The definitions of the application and its protocol(s) can be separated further by defining them in separate source files to be loaded at runtime.

In the the following example, a simple application may exhibit complex behavior at runtime, without its implementation itself being complex. The definition of protocol `bigtask` can be determined later, and managed separately.

```
char * load_pdl(char file_path) {
    /* omitted */
}
void main() {
    Arc_ProtocolDescription * pd = load_pdl("bigtask.pdl");
    Connector * c = connector_new(pd);
    PortId p0;
    connector_add_net_port(c, &p0,
       (FfiSocketAddr) {{127, 0, 0, 1}, 8000},
        Polarity_Putter, EndpointPolarity_Active);
    /*  [main]
           p0 --------------> (network) */
    connector_add_component(c, "bigtask", &p0, 1);
    /*  [main]   [bigtask]
                   p0 -----> (network) */
    connector_connect(c, -1);
}
```

The protocol description language is designed for symbiosis with the internals of the connector runtime. As such, it is able to express complex branching behavior intuitively as branching *speculations*. The language is designed to afford decision-making as a function of synchronized data before it is committed, while containing the resulting effects safely, such that the runtime's guarantees are preserved. For example, the following protocol determines how to route a given message synchronously, based on the message contents.

```
primitive route_by_contents(in a, out longer, out shorter) {
    while(true) synchronous {
        if(fires(a)) {
            msg m = get(a);
            if(m.length > 16) put(longer,  m);
            else              put(shorter, m);
        }
    }
}
```

In this fashion, protocol components can express a greater class of transactional interactions, whose behavior is unfolded lazily by the connector runtime as needed, and where transient violations of protocols are safely contained, and not leaked to the applications.

For example, consider this two-party consensus protocol, which enforces synchronous receipt of two equivalent messages on its inputs. This protocol can be used for two mutually-untrusting peers who wish to check whether they share a secret. While the protocol component is able to inspect the incoming messages speculatively, no causally related actions can succeed

if the assertion fails, preventing any information leaking to applications. In this particular case, the result is a kind of 'backpressure' on mismatching secrets, preventing the messages from being sent at all, such that neither is revealed.

```
primitive msg_consensus(in a, in b) {
    synchronous { assert(get(a) == get(b)) }
}
```

By relying on the predictable way a given protocol will affect the behavior of a communication session at runtime, protocols provide a rich and reliable expression of a session's safety properties. This allows for a modular approach to safety verification in large and complicated distributed systems, as the properties of protocols are inherently preserved regardless of their environment. Furthermore, as seen earlier, the application itself is alleviated of the burden of enforcing the needed requirements, instead being able to trust the connector itself to preserve the session's protocol once it is configured.

Consider how an application processing messages received through components instantiated with the following protocol can safely assume that all messages received have a length of 300 bytes.

```
primitive sync_300(in a, out b) {
    while(true) synchronous {
        msg m = get(a);
        assert(m.length == 300);
        put(b, m);
    }
}
```

Protocols are designed to make reasoning about communication behavior possible for machines as well as humans. Section 8.1 demonstrates how application developers can employ automated tools to apply circumstantial optimizations to their session's implementation in a fashion more just-in-time and verifiably safe than is possible for humans to do.

Care must be taken when writing protocols to bear in mind the causal dependencies the protocol descriptions express. The runtime won't find any solutions that require a message to be sent before it is received! The following `sync_eq` protocol consumes pairs of equivalent messages from inputs x and y and forwards them to its output z. Note that the way it is defined, the output is causally dependent on the receipt of a message through input x, as it precedes it in the control flow; this is not the case for y, which instead follows the output. In many sessions, this asymmetry between the inputs is not important. However, when instantiated in a session where x in turn depends on z (e.g., the output of z is synchronously routed back into x), the result is a cyclic causal dependency, prohibiting the realization of any behavior which requires x and z to transmit messages. Below, a simple session is demonstrated. The definition of this protocol component is provided below alongside a C application to instantiate it in a session which fails to transmit their message as the programmer might have expected.

```
primitive sync_eq(in x, in y, out z) {
    msg mx = null;
    msg my = null;
    while(true) synchronous {
        if(fires(z)) {
            mx = get(x);
            put(z, mx);
            my = get(y);
            assert(mx == my);
}   }   }
```

```
Arc_ProtocolDescription * pd = protocol_description_parse(pdl,
 ↪  pdl_len);
Connector * c = connector_new(pd);
PortId x, yp, yg, z;
connector_add_port_pair(c, &z,  &x ); // sync channel  z ->  x
connector_add_port_pair(c, &yp, &yg); // sync channel yp -> yg
connector_add_component(c, "sync_eq", 7, (PortId[]){x, yg, z});
connector_connect(c, -1);

connector_put_bytes(c, yp, "Hi", 2);
int code = connector_sync();
assert(code<0); // negative code indicates error: no solution found.
```

Intuitively, we may understand that there can be no solution in which *both* x and y causally depend on the output, we may wish to correct this asymmetry between `sync_eq`'s input ports. We can define a protocol component that finds solutions in which either of its inputs may be causally dependent on its output, not just x. This is simply achieved by injecting an additional speculative branch; effectively, we let the component speculate about which of its two input messages to receive first. In the current implementation, protocol components must express speculative branching in terms of port variables (Section 12.1.1 describes future work, aimed at removing this cumbersome limitation). As such, our component may employ a *dummy* port, whose only purpose is to facilitate the extra speculative fork. The new implementation for the `sync_eq` is provided below, complete with the relaxation of the causal relationship between x and z such that the previous example of a user's session would result in a successful synchronous round.

```
composite eq(in x, in y, out z) {
    // hides the creation of the dummy port
    channel dummyo -> dummyi;
    new eq_inner(x, y, z, dummyo, dummyi);
}
primitive eq_inner(in x, in y, out z, out dummyo, in dummyi) {
    msg ma = null;
    msg mb = null;
    while(true) synchronous {
        if(fires(z)) {
            if(fires(dummyi)) {
                // Dummy fires: x first
                mx = get(x);
                put(z, mx);
                my = get(y);

                // use dummy to avoid inconsistency
                put(dummyo, ma);
                get(dummyi);
            } else {
                // Dummy silent: y first
                my = get(y);
                put(z, my);
                mx = get(x);
            }
            assert(mx == my);
}   }   }
```

10.2 Canonical Reo Protocols in PDL

Reowolf's PDL is largely based on the Reo coordination language. Existing literature provides a comprehensive look at Reo's design philosophy at high level [Arb11, Arb16], and the details of the language [Arb04]. Note that in abstract applications, Reo protocols are often shown visually. For this work, to make for a simpler comparison with PDL, Reo protocols are defined in *treo*, [DA18] Reo's textual syntax.

This section compares and contrasts the two languages, and demonstrates how PDL is able to encode Reo circuits. In designing PDL, the intention was to facilitate cross-compilation between Reo and PDL, such that developments in one can carry over to the other.

10.2.1 Primitive Reo Connectors

Most essentially, Reo and PDL share the emphasis on compositional protocol definitions, such that complex protocols are defined in terms of simpler ones. The simplest protocols, which cannot be subdivided, are referred to as *primitive*, reflected in PDL by the `primitive` keyword.

In both languages, the most essential, primitive is the *sync* protocol, also referred to as the *sync channel* to reflect its primitive, end-to-end nature. They forward incoming messages to their output synchronously. They are idempotent, with individual instances interchangeable with arbitrarily long chains of sync channels without influencing the system's behavior.

Aside from superficial syntactic differences (E.g., treo marks the input/output polarity of ports with `?` and `!`, rather than `in` and `out` respectively), the most significant difference is Reo's declarative style, which is stateless by default, with each protocol defining a set of *guarded rules*, defining it's behavior during each synchronous round. Conversely, PDL draws inspiration from imperative languages (such as C) for (in addition to superficial syntax such as curly-braced block expressions, and typed variable declarations), making use of an implicit control flow top to bottom, overridden by conventional control flow statements such as `while` and `if`. Furthermore, Reo's syntax draws more direct inspiration from its roots in constraints on timed data streams (for more information, see [Arb04]), such that PDL's variable assignment corresponds to Reo's equality constraint, both of which are rendered syntactically as `=`. Consider the similarities and differences in how both languages define the sync protocol, which endlessly forwards an incoming message to the output if it exists.

```
sync(a?, b!){
    #RBA
    {a,b} a=b
}
```

```
primitive sync(in i, out o) {
    while(true) synchronous {
        if(fires(i)) put(o, get(i));
    }
}
```

The *lossy* channel is similar to the sync channel, but for each incoming message, makes a binary nondeterministic choice of whether the message is forwarded or lost. In the case of Reo, this behavioral branch is made explicit as the inclusion of two rules, the former guarded by the condition messages flow through both ports, resulting in the message being forwarded, and the latter guarded by the condition that a message flows through the input, but not through the output. In the case of PDL, this behavioral branch is the result of reflecting on `fires(o)`, a speculative boolean variable whose value is determined by the environment at runtime, whereafter the control flow diverges to include or exclude the consequent of the `if` statement.

```
lossy(a?, b!){
    #RBA
    {a, ~b} true
    {a, b} a=b
}
```

```
primitive lossy(in i, out o) {
    while(true) synchronous {
        if(fires(i)) {
            msg m = get(i);
            if(fires(o)) put(o, m);
        }
    }
}
```

Reo's syntax prioritizes brevity, enabling users to reason about the values of data, whose corresponding message-flow behavior includes the implicit replication of messages as necessary, which, in the case of two output ports, is equivalent to the insertion of the the `replicator2` component shown below. By contrast, PDL enforces the strict, unique ownership of ports, requiring such replication behavior to be expressed explicitly, as routed through a `replicator2` component.

```
replicator2(a?, b!, c!){
    #RBA
    {a, b, c} a=b=c
}
```

```
primitive replicator2(in a, out b, out c) {
    while(true) synchronous {
        if(fires(a)) {
            msg m = get(a);
            put(b, m);
            put(c, m);
        }
    }
}
```

Similarly, Reo messages routed through an output port from multiple input ports are implicitly merged, as if routed through an intermediary component, instantiated with `merger2`. As before, PDL has no such implicit behavior, requiring the use of a `merger2` component to explicitly merge the messages of two input ports into an output port.

```
merger2(a?, b?, c!){
    #RBA
    {a, b} a=b
    {a, c} a=c
}
```

```
primitive merger2(in a, in b, out c) {
    while(true) synchronous {
        if      (fires(b)) put(b, get(a));
        else if(fires(c)) put(c, get(a));
    }
}
```

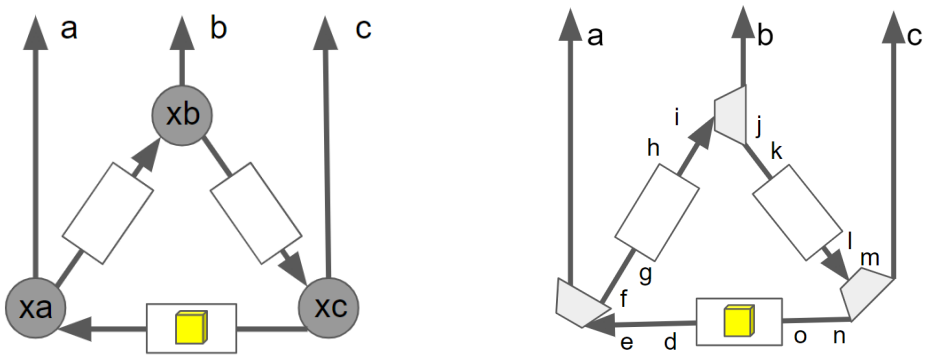
The *fifo1* channel is canonical for forwarding messages asynchronously through the use of a single buffer slot. The storage of a value facilitates a meaningful persistence of states between synchronous interactions; as such, fifo1 connectors are often used as the building block for more complex, stateful protocols.

In this case, Reo's manipulation of state is made explicit as reads and writes of a memory cell identified as `m`. Where the distinction between the 'new' value of a mutated cell must be distinguished from 'old' values, the identifier is suffixed by an apostrope `'`, once again, drawing inspiration from timed data streams, where identifiers can look ahead to values at the 'next' timestamp using `'`.

As is typical for imperative languages, a protocol's state in PDL changes implicitly with the control flow, and explicitly with mutations of local variables. In the case of the fifo1 connector, the only meaingful state to persist between synchronous rounds is the value of variable `m`, which serves to store either one message, or `null`.

```
fifo1(a?, b!) {
    #RBA
    {a, ~b} $m = null, $m' = a
    {~a, b} $m != null, b = $m, $m' = null
}
```

```
primitive fifo1(in a, out b) {
    msg m = null;
    while(true) synchronous {
        if(m == null) {
            if(fires(a)) m = get(a);
        } else {
            if(fires(b)) put(b, m);
            m = null;
        }
    }
}
```



(a) Reo

(b) Visual PDL

Figure 10.1: Visual representation of the composite sequencer3 protocol, as expressed by treo and PDL respectively.
(a) Vertices are port nodes. rectangles over arrows are `fifo1` or `fifo1_full` components. Dangling arrows are boundary ports.
(b) Vertices are components, whose shapes correspond to their protocol: rectangles are `fifo1` or `fifo1_full`, trapezia are `replicator`. Arrows are channels with one or two ports on either end.

10.2.2 Composite Reo Connectors

Some protocols occur often enough that they are canonical to Reo, despite not being primitive.

The first example is the *sequencer*, which emits *signals* (messages with no contents) out of its output ports, one at a time, in a cyclical fashion. For simplicity, we focus on the `sequencer3`, for which the period is three in particular. Both languages achieve this by three fifo1 channels cycling a single token message. Each time the token moves, a replica of the token is emitted by one of the three output channels, starting with port `a`. The state of this protocol is contained entirely by its fifo1 channels, one of which begins filled with a signal message.

PDL renders this protocol as a composition of six primitive protocols, three fifo1 channels, and three *replicators*. These primitives are connected into a ring by six freshly-created sync channels, each exposed as an input and ouput port pair. This explicit rendering of PDL places the burden of preserving the linear ownership of ports by protocol components onto the programmer. This is depicted visually in Figure 10.1b. This representation expedites interpretation at runtime, by preserving the linearity of ports from their creation to consumption by a child protocol. For example, `g` is created in line `channel f -> g;` and moved in line `new fifo1(g, h);`.

Reo's rendering is simplified through the use of implicit replication for data flow from identified nodes to multiple outputs. Otherwise, its representation of the protocol is much the same.

```
sequencer3(a!, b!, c!) {
    sync(xa, a)
    sync(xb, b)
    sync(xc, c)
    fifo1full(xc, xa)
    fifo1(xa, xb)
    fifo1(xb, xc)
}
```

```
composite sequencer3(out a, out b, out c) {
    channel d -> e;
    channel f -> g;
    channel h -> i;
    channel j -> k;
    channel l -> m;
    channel n -> o;

    new fifo1_full(o, d);
    new fifo1(g, h);
    new fifo1(k, l);
    new replicator(e, f, a);
    new replicator(i, j, b);
    new replicator(m, n, c);
}
```

Our final example is `xrouter`, which routes incoming messages to one of two output ports non-deterministically.

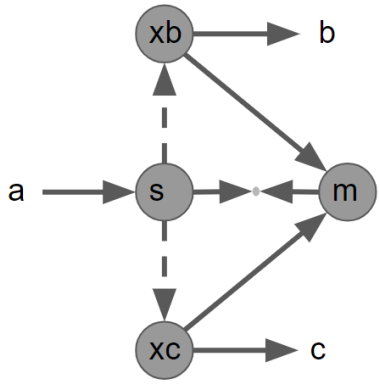
Figure 10.2 shows the protocol's Reo definition graphically. All consistent protocols are all primitives: lossy synchronous channel (dashed arrow), synchronous channel (single unbroken arrow) and synchronous drain (two colliding arrows). The protocol works by constraining the flow of messages such that only the desired behavior is possible while satisfying all constituent protocols: whenever a message arrives from `a`, a message is emitted from `b` or from `c` but not both. The synchronous drain channel between `s` and `m` can be understood as expressing 'a message flows through `s` if and only if a message flows through `m`', effectively ensuring *at least* one of the outputs will emit a message, while node `m` is defined to include an implicit non-deterministic merger of messages from `xb` and `xc`, effectively ensuring that *at most* one of the outputs will emit a message.

The expression of this protocol in PDL corresponds closely to that of textual Reo, but as is the case for `sequencer3`, must bear the burden of explicitly expressing components expressed implicitly by Reo's nodes. Reo nodes `xb`, `s`, and `xc` all make use of implicit replication, corresponding to instances of `replicator2` in the PDL definition. Furthermore, `m` makes an implicit use of a non-deterministic merger of multiple incoming channels, corresponding to an instance of `merger2` in the PDL definition.

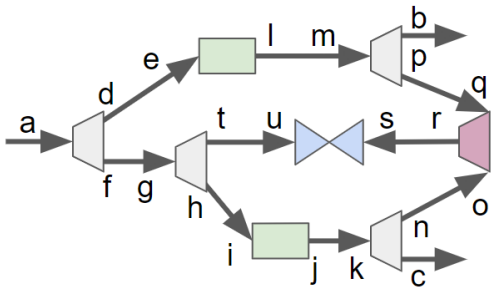
```
syncdrain(a?, b?) {
    #RBA
    {a, b} true
}
xrouter(a!, b!, c!) {
    sync(a, s)
    sync(xb, b)
    sync(xc, c)
    lossy(s, xb)
    lossy(s, xc)
    sync(xb, m)
    sync(xc, m)
    syncdrain(s, m)
}
```

```
primitive sync_drain(in a, in b) {
    while(true) synchronous {
        if(fires(a)) {
            get(a);
            get(b);
        }
    }
}
composite xrouter(in a, out b, out c) {
    channel d -> e;
    channel f -> g;
    channel h -> i;
    channel j -> k;
    channel l -> m;
    channel n -> o;
    channel p -> q;
    channel r -> s;
    channel t -> u;

    new lossy(e, l);
    new lossy(i, j);
    new sync_drain(u, s);
    new replicator(a, d, f);
    new replicator(g, t, h);
    new replicator(m, b, p);
    new replicator(k, n, c);
    new merger(q, o, r);
}
```



(a) Reo



(b) Visualized PDL

Figure 10.2: Visual representation of the composite xrouter protocol, as expressed by treo and PDL respectively.
(a) Vertices are port nodes. rectangles over arrows are `fifo1` or `fifo1_full` components. Dangling arrows are boundary ports.
(b) Vertices are components, whose shapes correspond to their protocol: rectangles are `lossy_sync`, trapezia are `replicator` when gray and `merger` when red, and the bowtie is a `sync_drain`. Arrows are channels with one or two ports on either end.

10.3 Connector-Socket Inter-Operability

This section compares the usages of connectors and sockets for exchanging UDP datagrams over the network. This includes the use of both the connector API and pseudo-socket API for connectors. For the sake of brevity, this section includes examples are chosen to demonstrate their message exchange funtionality, and as such, they do very little else. Realistic usages will often be more sophisticated, but their interaction with the (pseudo-)sockets is essentially the same, while being otherwise cluttered with details irrelevant to the demonstrations, such as manipulations of the contents of datagrams. For example, the pseudo-socket API can support the implementation of a DNS client whose usages of the pseudo-socket boil down to a sequence of procedure calls similar to those shown in the examples to follow.

To begin, we consider a simple C program which uses conventional UDP sockets in blocking mode to receive a single datagram, and print the contents of the received messages as a hexadecimal string. The UDP socket is used in a connection-oriented fashion, locally bound to localhost port 8000, and communicating with a peer at localhost port 8001:

```
#include <netinet/in.h> // defines socketaddr_in
#include <stdio.h>  // defines printf
#include <stdlib.h> // defines malloc, free
#include <unistd.h> // defines close
#include <arpa/inet.h> // defines inet_pton
#define BUFSIZE 128
int main() {
    // --- setup ---
    struct sockaddr_in addrs[2];
```

```
    addrs[0].sin_family = AF_INET;
    addrs[0].sin_port = htons(8000);
    inet_pton(AF_INET, "127.0.0.1", &addrs[0].sin_addr.s_addr);
    addrs[1].sin_family = AF_INET;
    addrs[1].sin_port = htons(8001);
    inet_pton(AF_INET, "127.0.0.1", &addrs[1].sin_addr.s_addr);
    int fd = socket(AF_INET, SOCK_DGRAM, 0);
    bind(fd, (const struct sockaddr *)&addrs[0], sizeof(addrs[0]));
    connect(fd, (const struct sockaddr *)&addrs[1],
    ↪  sizeof(addrs[1]));

    // --- communication ---
    char * buffer = malloc(BUFSIZE);
    size_t msglen, i;
    msglen = recv(fd, (void *)buffer, BUFSIZE, 0);
    for(i=0; i<msglen; i++) {
        printf("%02X", buffer[i]);
    }

    // --- cleanup ---
    close(fd);
    free(buffer);
    return 0;
}
```

The same program is shown below, re-implemented in terms of the pseudo-socket API. In this case, no macros are employed to hide the injection of the `rw_` prefix on the identifiers of pseudo-socket procedures. Otherwise, the only difference from the program above is the inclusion of the pseudo-socket header file, whose contents are provided in Section 9.2.3.

```
#include <netinet/in.h> // definies socketaddr_in
#include <stdio.h>  // defines printf
#include <stdlib.h> // defines malloc, free
#include <unistd.h> // defines close
#include <arpa/inet.h> // defines inet_pton
#include "pseudo_socket.h"
#define BUFSIZE 128
int main() {
    // --- setup ---
    struct sockaddr_in addrs[2];
    addrs[0].sin_family = AF_INET;
    addrs[0].sin_port = htons(8000);
    inet_pton(AF_INET, "127.0.0.1", &addrs[0].sin_addr.s_addr);
    addrs[1].sin_family = AF_INET;
    addrs[1].sin_port = htons(8001);
    inet_pton(AF_INET, "127.0.0.1", &addrs[1].sin_addr.s_addr);
    int fd = rw_socket(AF_INET, SOCK_DGRAM, 0);
    rw_bind(fd, (const struct sockaddr *)&addrs[0],
    ↪  sizeof(addrs[0]));
```

```
    rw_connect(fd, (const struct sockaddr *)&addrs[1],
    ↪  sizeof(addrs[1]));

    // --- communication ---
    char * buffer = malloc(BUFSIZE);
    size_t msglen, i;
    msglen = rw_recv(fd, (void *)buffer, BUFSIZE, 0);
    for(i=0; i<msglen; i++) {
        printf("%02X", buffer[i]);
    }

    // --- cleanup ---
    rw_close(fd);
    free(buffer);
    return 0;
}
```

Finally, the following is an example of a usage of the connector API to communicate with both of the prior programs through the use of two UDP mediator components. To avoid duplicate bindings, the pseudo-socket and connector API programs use a UDP channel whose endpoints are bound to localhost 8002 and 8003 respectively. In the resulting system, the sending of the two user messages `Hello, socket!` and `Hello, pseudo-socket!` occur synchronously, but the synchrony does not propagate through each message's respective UDP channel to the recipients.

```
#include "reowolf.h"
int main() {
  // --- setup ---
  Arc_ProtocolDescription * pd = protocol_description_parse("", 0);
  Connector * c = connector_new(pd);
  PortId putter_a, putter_b;
  FfiSocketAddr addresses[4] = {
    {{127, 0, 0, 1}, 8000},
    {{127, 0, 0, 1}, 8001},
    {{127, 0, 0, 1}, 8002},
    {{127, 0, 0, 1}, 8003},
  };

  // putter_a to UDP mediator (getter id discarded)
  // bound to addresses[0], connected to addresses[1]
  connector_add_udp_mediator_component(c, &putter_a, NULL,
  ↪  addresses[1], addresses[0]);
  // putter_b to UDP mediator (getter id discarded)
  // bound to addresses[2], connected to addresses[3]
  connector_add_udp_mediator_component(c, &putter_b, NULL,
  ↪  addresses[3], addresses[2]);
  connector_connect(c, -1);

  // --- communication ---
  // synchronous put of 14- and 21-byte messages
```

```
  connector_put_bytes(c, putter_a, "Hello, socket!", 14);
  connector_put_bytes(c, putter_b, "Hello, pseudo-socket!", 21);
  connector_sync(c, -1);

  // --- cleanup ---
  protocol_description_destroy(pd);
  connector_destroy(c);
  return 0;
}
```

Bibliography

[Arb04] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(3):329–366, 2004.

[Arb11] Farhad Arbab. Puff, the magic protocol. In *Formal Modeling: Actors, Open Systems, Biological Systems*, pages 169–206. Springer, 2011.

[Arb16] Farhad Arbab. Proper protocol. In *Theory and Practice of Formal Methods*, pages 65–87. Springer, 2016.

[DA18] Kasper Dokter and Farhad Arbab. Treo: Textual syntax for reo connectors. *arXiv preprint arXiv:1806.09852*, 2018.

Chapter 11

Performance Benchmarks

This chapter complements the usage and examples of Chapter 10 by facilitating a more thorough understanding of the way connectors work. We explore the way the properties of the user's session affect the connector runtime implementation. Of primary focus is the performance of the *synchronization procedure*, exposed to the user as `connector_sync` in the connector API for driving the session's communications. This procedure is involved with every facet of the implementation, and is most likely to be performance-critical to application developers. The chapter also serves to complement the descriptions of previous chapters, including the design of connectors (Chapters 2) and the explanation of the implementation (Chapter 6) with concrete examples as experimental test cases. Wherever possible, redundancy is kept to a minimum by re-using protocols presented in Chapter 10. The observations resulting from the benchmarks motivate directions for future work, presented in Chapter 12.

The measurements represented in this chapter are the result of measurements of runtime on a the *test machine*, whose technical specifications are laid out in Table 11.1. All figures to follow show measurements of runs for connectors setup, and then completing N synchronization rounds before exiting; runtimes reflect the duration elapsed between the start of the first and end of the last round. N is chosen on a case-by-case basis to result in a total runtime between 1 and 120 seconds, and $N \geq 1000$, for all non-trivial benchmarks (large enough to gather a good sample size, small enough to expedite the testing process); for example, all figures up to and including Figure 11.7 as they appear chronologically, were run with $N = 10^6$. Tests with significant variability in runtime between runs (e.g. those involving real networks) were entirely repeated 3–5 times, with each measurement either shown in the figure as a distinct measurement (with a $\times$ symbol) or represented in the plot as the mean value.

Architecture	Intel x86 64-bit
Operating System	Windows 10 Pro (2004) build 19041.508
Processor	Intel i7-7700 @ 4.2 GHz with 4 physical / 8 logical cores. Caches L1: 8x32KB, L2: 4x256KB, L3: 8MB
Memory	16GB DDR4 @ 3200MT/s
Storage	500GB (512MB cache) 3d v-nand (TLC) NVMe SSD with interface 4x PCI Express 3.0

Table 11.1: Technical specification of the test machine, used in this section for performance benchmarking.

Chapter 10 starts with the simplest possible sessions, systematically complicating the configuration to explore the properties of the session's configuration on its runtime. Section 11.2 exemplifies the utility of session optimization, introduced in Section 8.1.

11.1 Baseline Synchronization Performance

This section benchmarks the performance of the connector's `connector_sync` procedure, as it drives the message exchange, as well as the distributed control algorithms that facilitate it.

By design, connectors have an immense configuration space, making it infeasible to perform a sufficiently diverse set of benchmarks such that the performance of any realistic use case is represented. Instead, our approach teases out the performance characteristics of `connector_sync` as a function of the most essential properties of the configuration, such that one can form an understanding of how the essential properties of any connector configuration influences the runtime performance of the connector.

11.1.1 Configuration Complexity

Before introducing the overhead that follows from management of a distributed system, this section explores the runtime overhead of the connector's synchronization procedure as a function of its session configuration.

Trivial Synchronization Procedure

The most essential baseline benchmark measures the cost of a `connector_sync` with the most trivial configuration, which exchanges an empty set of messages. As the runtime is not optimized to check for this degenerate case, the synchronization procedure prepares all of the internal bookkeeping state in preparation for a consensus procedure which is immediately resolved. The following code snippet in the C programming language shows the experimental setup for this test, which shares its essential structure with all the experiments to follow. On the test system, this program finishes in 1.225 seconds, allowing us to conclude that the mean duration of a single `connector_sync` was 1.225 μs. This result is very consistent between runs, in part owing to the computation being deterministic, except for the management of underlying system resources (e.g. memory layout) which plays a small enough role not to influence the result.

```c
#include <time.h>
#include "reowolf.h"
int main(int argc, char** argv) {
    Arc_ProtocolDescription * pd = protocol_description_parse("", 0);
    Connector * c = connector_new(pd);
    connector_connect(c, -1);
    clock_t begin = clock();
    int i;
    for (i=0; i<1000000; i++) {
        connector_sync(c, -1);
    }
    clock_t end = clock();
    double time_taken = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Time taken: %f\n", time_taken);
    return 0;
}
```

Ports

Next, we establish the cost of maintaining a set of N unused ports. Concretely, the previous test is repeated, but with addition of ports into the interface of the protocol component. This experiment was repeated with ports created in either of the two ways available to the programmer: (a) as a pair of ports, connected through a memory channel, or (b) a pair of individual ports, forming two halves of a network channel over the transport-layer. The results of this test are shown in Figure 11.1, plotted over the number of silent interface ports.

We see that the runtime duration in microseconds R of `connector_sync` is a good fit for the linear function of the number of interface ports P given by $R = 1.2 + 0.13 \cdot P$, shown in the figure in solid gray. Clearly, there is not a significant difference between the ports with and without an underlying network channel. This linear cost is explained by the relationship between a component's interface port set, and the *solution* which encodes the successful result of the round, which ultimately specifies constraints for all the system's ports, in this case, predicating silence.

Protocol Components

Recall that application developers are able to instantiate protocol components, delegating the task of maintaining their states to the connector runtime. This test establishes the baseline cost of maintaining a set of N identically protocol components, each owning zero ports, and doing no work other than (trivially) participating in each synchronous round. Without including a session optimization step, the runtime works to retain the states of these components, despite them having no effect on the system's observable behavior. The behavior of these components is given by the following protocol, expressed in PDL:

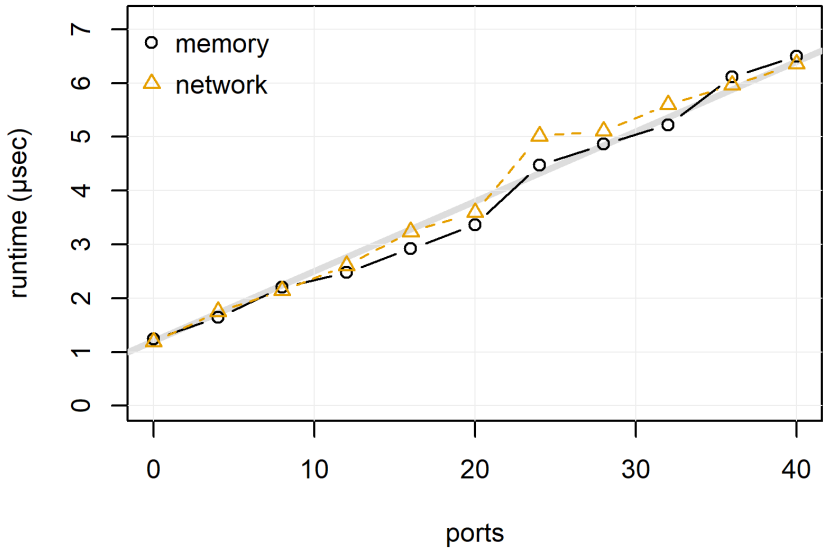


Figure 11.1: Runtime of `connector_sync` as a function of the number of silent ports in an otherwise trivial connector configuration. Lines distinguish measurements for runs where ports are created with underlying memory channels, or transport-layer network channels. The gray underline illustrates a linear fit.

```
primitive trivial_loop() {
    while(true) synchronous{}
}
```

The runtime performance of `connector_sync` is plotted over the number of these trivial protocol components in Figure 11.2. In all cases, such protocol components are more expensive to maintain than the same number of silent ports. It comes as no surprise that components are at least as expensive as ports, as both encumber the discovery of a solution in a similar manner; the solution constrains the message flowing through each ports, and a solution necessitates the composition of a set of predicates, one from each component.

The relationship in the figure appears to be almost linear, but becoming superlinear in excess of approximately 13 components. The added cost is explained by the significantly larger and more complex storage structures necessary to maintain these protocol components. The superlinear relationship is explained by this storage growing to the extent that the machine makes less effective use of caching.

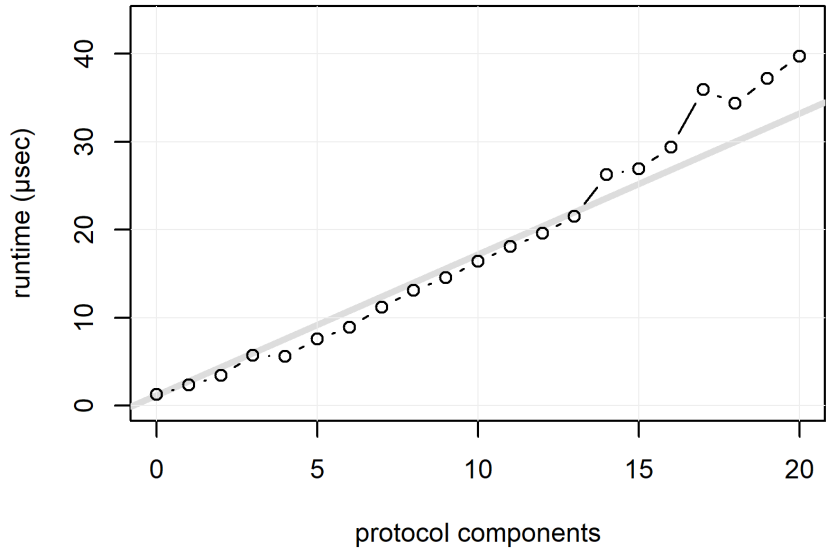


Figure 11.2: Runtime of `connector_sync` as a function of the number of trivial protocol components in an otherwise trivial connector configuration. The gray underline shows a linear extrapolation fitting the leftmost samples to clarify the threshold beyond which the relationship is superlinear.

Protocol Component Interpretation

Protocol components perform work between and during synchronous rounds in accordance with their protocol specification. This next experiment attempts to quantify the cost of adding computation work to the definition of the instantiated protocol components to be performed at runtime by the connector runtime's interpreter. The protocol components used are defined in the following PDL description, differing from the definition of the previous trivial protocol descriptions by the addition of a (small) sequence of operations that read and write the component's local variable store, and jumps in control flow.

```
primitive presync_work() {
    int i = 0;
    while(true) {
        i = 0;
        while(i < 2) i++;
        synchronous {}
    }
}
```

The prior experiment is repeated, comparing the runtimes with `presync_work` components with their `trivial` counterparts in Figure 11.3. We see that the new components consistently result in significantly more expensive synchronization, despite the new components performing only a very small local task each synchronous round. We conclude that the cost of interpretation of these local memory manipulations has a considerable impact on performance. Furthermore, we see some chaotic stratification of runtimes into two distinct bands. Runs with the exact same initial configuration can be seen to fall into either stratum unpredictably; this stratification being observable through each measurement representing the cumulative runtime of a million runs suggest that whatever property determines the runtime persists for the duration of the run. As the synchronization work is still entirely deterministic, the most likely candidate is the memory layout of the protocol component's storage resulting in more or less effective use of caching.

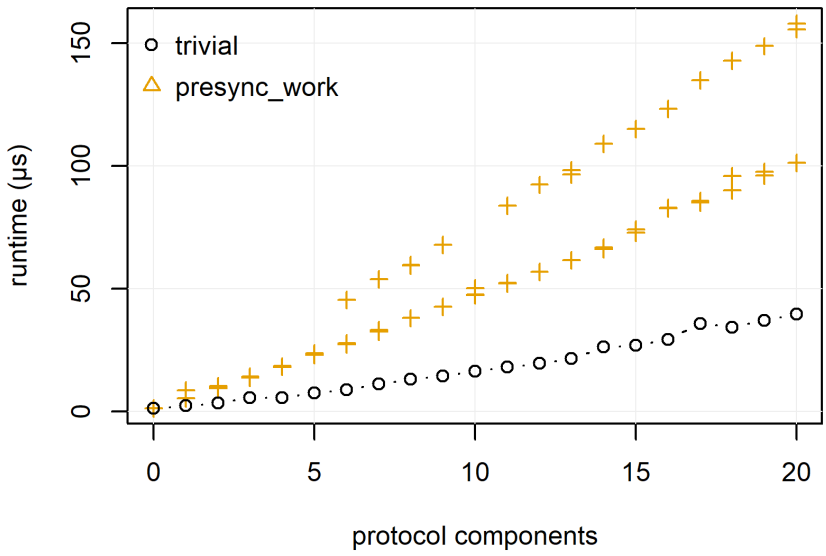


Figure 11.3: Runtime of `connector_sync` as a function of the number of protocol components in an otherwise trivial connector configuration. Runtimes are distinguished by the protocol used to define the components.

Ports and Components Together

The previous experiments teased out the relationships of ports or components to the duration of the synchronization procedure independently. This section examines the effects of combining the inclusion of ports and trivial components in the same run. Figure 11.4 gives two views on runtimes measured from combinations of the two parameters in question. As before, components are consistently more costly to maintain than an equivalent number of ports. However, the figure shows that configurations with many ports and many components can expect to be at

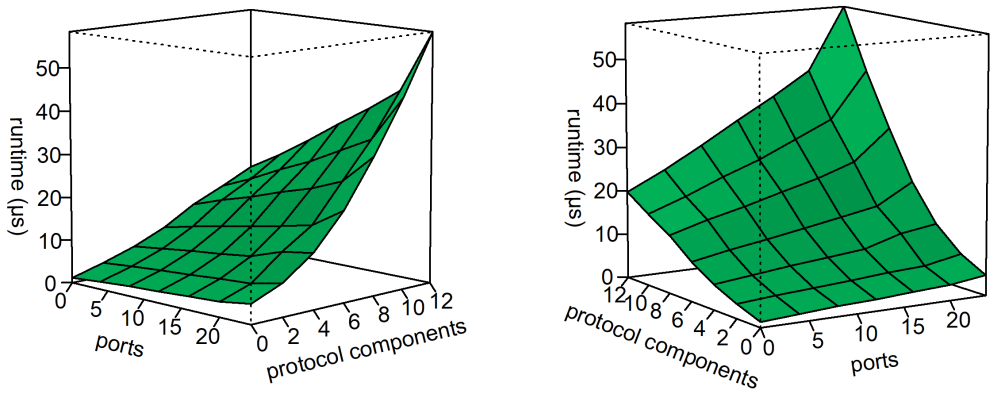


Figure 11.4: Runtime of `connector_sync` as a function of the number of silent ports and trivial protocol components in an otherwise trivial connector configuration.

least nominally more expensive than the sum of their parts. Figure A.1 in the appendix provides another view of these measurements as a heatmap.

User Message Routing

The experiments from this point on require messages to be exchanged between components. As a first step, we measure the performance cost of the cheapest form of message routing: routing through in-memory channels.

This experiment involves a connector always configured with 13 `sync` protocol components, and 28 ports, such that every component (including the native component) owns precisely one input and one output port. The test routes the message `"Hello, world!"` through a ring of components, starting and ending with the native component, through a chain of N sync components, while the other $13 - N$ sync components excluded from the ring, thus routing no message.

Figure 11.5 shows that while all runs have a baseline cost in common to maintain the same set of components (shown as the area under the gray line), the runtime cost of routing the message through the ring scales linearly with the length of the ring. Note that this includes the cost of the routing itself, as well as the cost of mutating the states of protocol components that send and receive the message, which is a cost that can unfortunately not be isolated.

Native Component Speculation

Recall that all components have some power to speculate, expressing the results of nondeterministic choice, relying on the runtime to prune conflicting options, and decide the outcome such

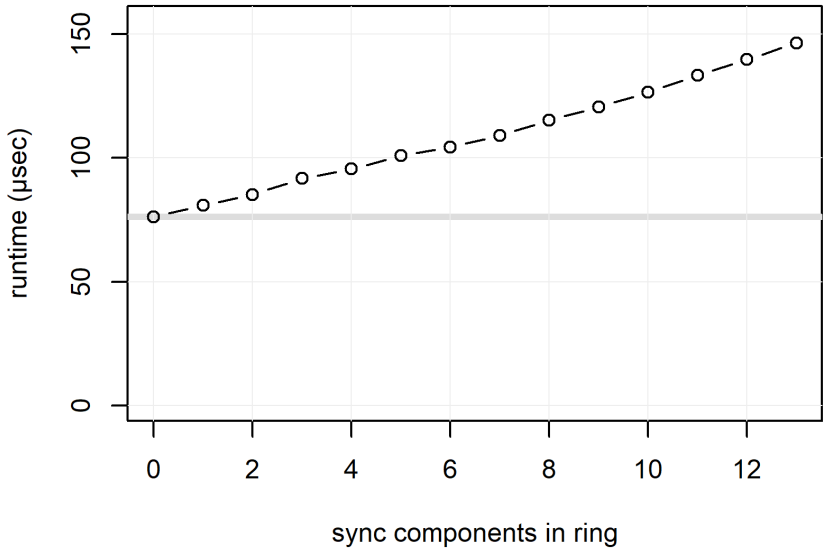


Figure 11.5: Runtime of `connector_sync` as a function of the number of sync components included in the ring through which a single message is sent, starting and ending at the native component. In all cases, the connector is configured to include 13 sync components; runs differ only in the way components are connected to one another.

that all components' constraints are satisfied. Native components (the role the user application plays in the communication session) can distinguish mutually exclusive options as *batches*, incrementally building them with repeated invocations of procedures in the connector API.

The connector runtime drives the search for satisfactory solutions by searching for consistent combinations of partial solutions, contributed by components. In the case of native components, these local solutions correspond one-to-one with batches, and as they cannot be causally dependent on anything else, the runtime starts by eagerly unfolding them before the main synchronization loop begins.

This test measures the performance cost of `connector_sync` as a function of the number of batches, with which the number of connector API calls, and batches processed eagerly by the runtime scales linearly. The results shown in Figure 11.6 confirm this expectation, showing linear growth with the number of batches above a constant, which represents the work constant over the number of batches, shown with a gray guideline.

Protocol Component Speculation

Each protocol components can speculate too, searching the space of potential local solutions. However, the connector runtime unfolds their speculative computation lazily, in response to the availability of new speculative information. This process can be understood as lazily unfolding

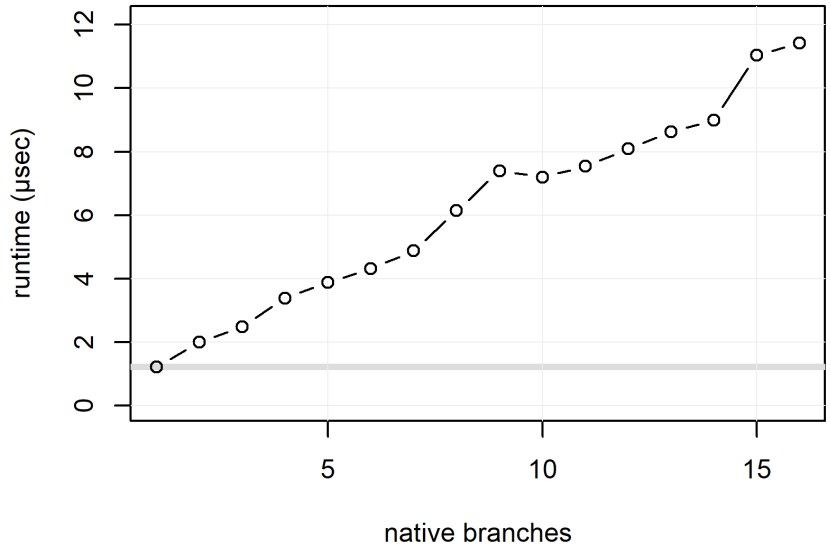


Figure 11.6: Runtime of `connector_sync` as a function of the number of speculative branches the native component explores.

the exponential domain of possible solutions in response to message exchanges that may result in a component finishing the round.

This test presents a protocol configuration designed to force the speculation of solutions to unfold the options as far as possible, resulting in the runtime managing a growing set of potential solution combinations. Every round succeeds with the native component sending a message whose payload is a single null byte through a chain of `forward` components to a `recv_zero` component on the end. This component's definition is provided in PDL below, and can be understood simply as a component that always enforces the receipt of a message containing a single null byte through its only port:

```
primitive recv_zero(in a) {
    while(true) synchronous {
        msg m = get(a);
        assert(m[0] == 0);
    }
}
```

The trick is in the native component offering a set of *M* potential messages through the chain, all but one of which the `recv_zero` component will ultimately reject, but not before each is transmitted through the chain of *N* `forward` components, creating, for each, a new speculation about the message flowing through its input and output ports. To control for the order of these

options being a factor, each run represents the index of the 'correct' choice cycled, such that the choice of index is evenly distributed over the options.

Figure 11.7 shows the runtime performance of this connector in response to different choices for parameters M and N. We expect runtime to scale with a value somewhere between N (if the cost of managing N components dominates) and $M * N$ (if the cost of managing the speculative branches and partial solutions dominates). What we see is something to that effect, with the usual gentle curve upward for runs requiring a larger memory footprint.

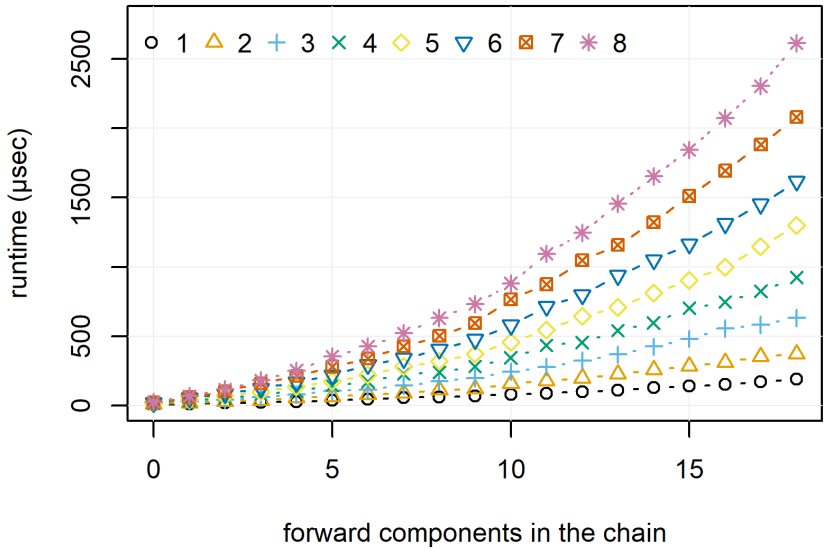


Figure 11.7: Runtime of `connector_sync` as a function of the length of the chain of `forward` components between the native component at the head, and the `recv_zero` component at the tail. Different lines show a different choice for the number of messages the native component offers, only one of which results in a solution.

11.1.2 Distributed Coordination

This section explores the overhead that follows from the connector runtime being meaningfully distributed over several hosts.

Multi-party Consensus

The moment several applications want to share a communication session, `connector_sync` must employ a non-trivial distributed consensus procedure. Section 6.2.3 describes how this is achieved by leveraging the *consensus tree*, a logical overlay network built during `connect`,

imposing a hierarchy on the participating connectors in the way they accumulate control information.

This test sets up the simplest possible case for a multi-party session: a set of N applications (each with its own process, and own connector structure) forming a continuous chain with the leader at one end, as depicted in Figure 11.8. For now, we concentrate on using *localhost* as the transport, representing the 'smallest' step up from local memory synchronization seen thus far. Figure 11.9 shows the runtime resulting from all connectors synchronizing without exchanging any messages. The runtime scales almost linearly with $(N - 1)$, as this represents the number of 'hops' in the chain, where each round results in a wave of consensus from the (single) leaf node, up to the leader, and a wave of acknowledgement back. The gray guideline aids in recognizing the relationship as marginally superlinear; as a result of control messages scaling with the number of ports in the session.

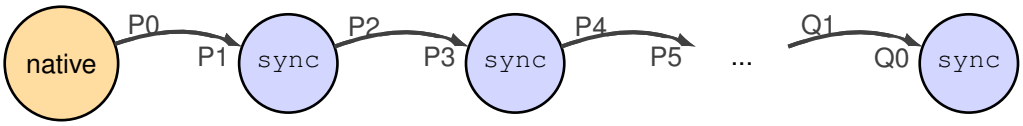


Figure 11.8: Component graph of a session with one native, connected such that it sends messages through a chain of `sync` components.

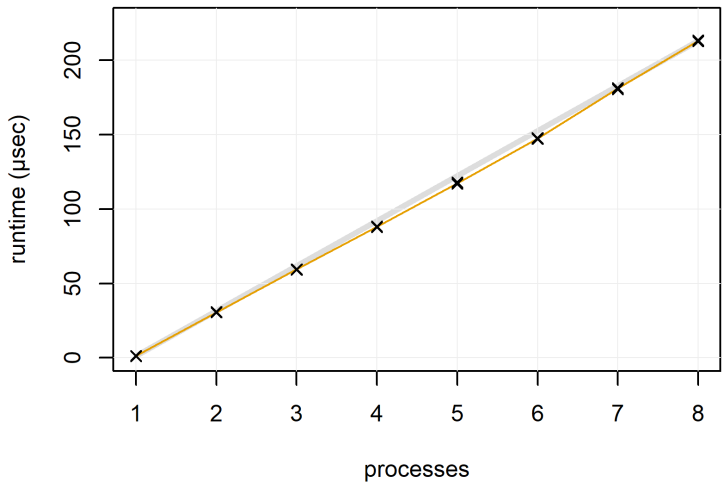


Figure 11.9: Runtime of the leader's `connector_sync` for a chain of N connectors connected by ports which transport messages over the localhost network, plotted over N. Runtimes are shown as crosses, with the line connecting the mean of each choice of N. The gray guidelines shows a linear interpolation for means of runtimes for runs with 1 and 8 processes.

Note that runtime of a session with participants linked in such a 'chain' with the leader at once end is the worst case scenario for the performance of the consensus algorithm, as the

wave of consensus must bounce from the end of the chain and back again, every synchronous round. Figure 11.11 plots the results of repeating the experiment with the chain connected with an extra network channel at the ends to form a ring, as depicted in Figure 11.10. Despite these runs having strictly more ports than in the case of the chain topology, we observe that in nearly all cases, the runtime has decreased, and that the duration of particular runs with the same configuration have diverged (where before they were very consistent). This is the result of the ring topology affording numerous constructions of the session's internal consensus tree, whose topology is determined by the result of control messages racing one another during the setup procedure. In all runtimes measured for 3 or more processes, the consensus tree has a shorter 'height' the number of hops from the root to its most distant leaf, where a smaller height corresponds to a more balanced tree. In these runs, where the networking latency delaying the consensus procedure predominates the overhead, there is a nearly linear relationship between the measured runtime, and the height of the consensus tree.

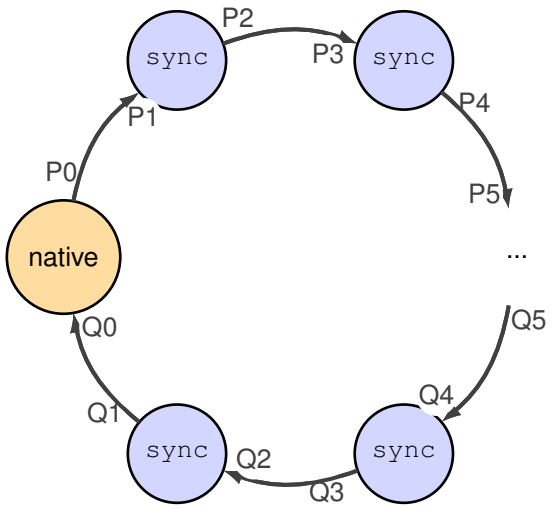


Figure 11.10: Component graph of a session with one native, connected in a ring of `sync` components.

Network Latency

When sessions span vast geographic regions, the completion of the distributed consensus procedure is inhibited by the latency of the underlying network transport. Fiture 11.12 shows the runtimes of a chain of *N* connectors reaching trivial consensus, i.e., the same configuration as that shown in Figure 11.9. In this case, the transport channels are configured to traverse the internet through the use of a proxy server; ultimately, IP packets are bounced between Amsterdam (Netherlands) and Les Escaldes (Andorra), resulting in channel links predicted by the `ping` network utility to require a round-trip time of of 15 milliseconds. We observe a linear relationship between the length of the chain and the mean duration of `connector_sync` once again, but this time at a significantly larger scale (note that runtimes are now measured in milliseconds), and with a chaotic stutter as a consequence of the chaotic fluctuations in latency introduced by the internet transport. As the latency of this internet transport is inherently noisy, we cannot draw any definitive conclusions about the contributions to runtime in this session; however, given

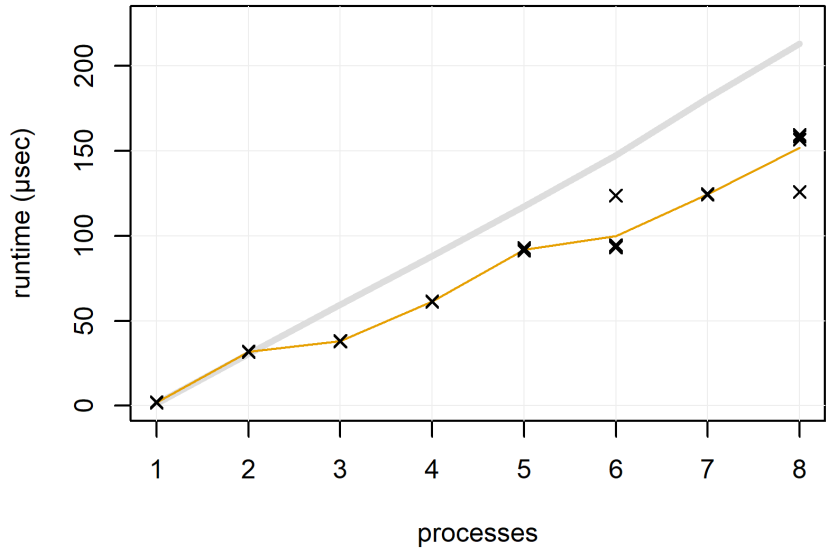


Figure 11.11: Runtime of the leader's `connector_sync` for a ring of N connectors connected by ports which transport messages over the localhost network, plotted over N. Runtimes are shown as crosses, with the colored line connecting the mean of each choice of N. The gray guideline compares these results to those shown in Figure 11.9.

the findings of the previous experiments, it seems safe to conclude that the runtimes measured here are predominated by the network latency.

Causal Dependency

The connector runtime works to decide valuations for the messages passing through ports such that all the configured protocol's constraints are satisfied. Previously, Section 11.1.1 explained that the runtime drives the speculation of these messages' contents lazily, unfolding the possibilities as information becomes available. More concretely, the runtime traces the *causal dependency* from the `get` actions on a port to the `put` action on its peer, on the other end of the channel. Effectively, instead of enumerating all the messages that might arrive at a port, the runtime will 'play out' the effects of receiving a message when concrete messages are speculatively sent. The cost of this approach is introducing a delay before the message received by a `get` action is determined. This has an effect on the runtime of the synchronization procedure.

We examine the test case depicted in Figure 11.13, which attempts to understand the contributions of causal dependencies between the actions that make up the interaction completed by `connector_sync`. Two connectors, A and B, form a shared session, separated by a high-latency connection over the internet (estimated by `ping` to have a round-trip time of 15 milliseconds). Every round, N_A, the native component of A, sends a set of X messages, ultimately

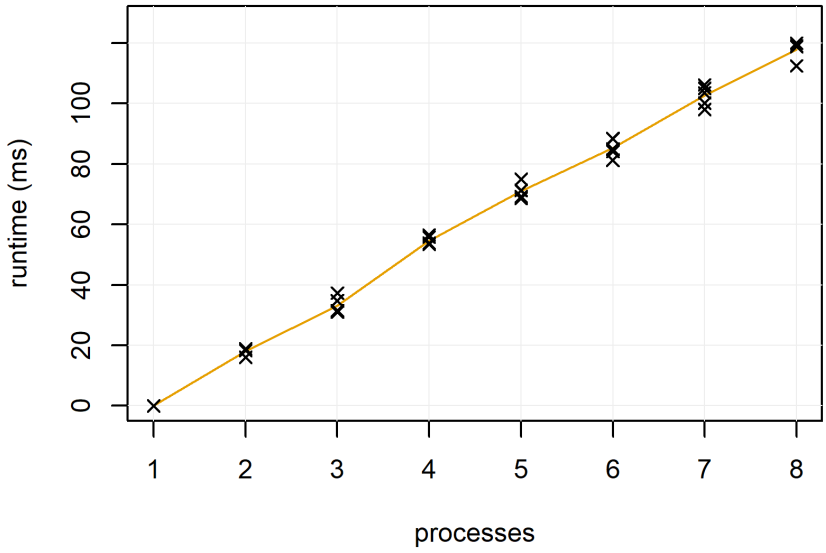


Figure 11.12: Runtime of the leader's `connector_sync` for a chain of N connectors connected by ports which transport messages over the internet between Les Escaldes and Amsterdam, plotted over N. Runtimes are shown as crosses, with the colored line connecting the mean of each choice of N.

received by N_B, the native component B. However, all messages are looped from follower to leader and back again before arriving at their destination Y times. This 'looping' of each message is achieved by the channel being routed through a pair of `forward` protocol components, who work to bounce incoming messages over the internet and back again.

Figure 11.14 shows the mean duration of the leader's `connector_sync` procedure for combinations of the parameters X and Y; note that these results feature noise, characteristic of runtimes subject to the ever-changing latency of a real internet connection. As was seen in Figure 11.4 previously, runtime consistently increases with large combinations of many ports and components, representing the overhead of managing more states, reasoning about a larger solution space, and transmitting larger control messages. The growth over the two input parameters make clear that sessions which include longer chains of causal dependency (having more loops) increase the time needed to complete the synchronous round, above and beyond a constant baseline duration of approximately 20 milliseconds to complete the synchronization procedure. Figure A.2 in the appendix provides another view of these measurements as a heatmap.

Message Size

Transporting message data between components is an important task for all communication protocols. This next experiment attempts to tease out the effect of the size of these messages

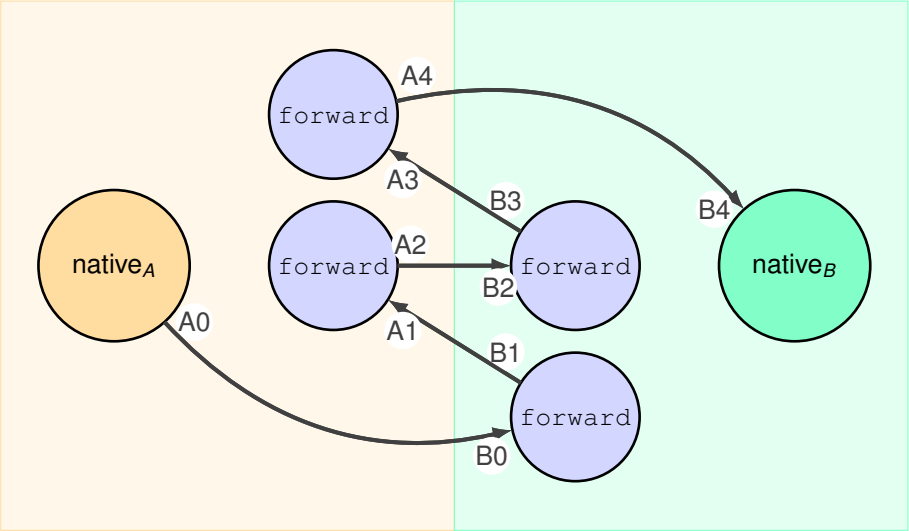


Figure 11.13: Component graph of a session with connectors A and B (left and right respectively), where the native component of the former sends X messages each round to the native component of the latter, via a chain of `forward` components distributed such that messages loop back from B to A to B again Y times. The figure depicts the case where $X = 1$ and $Y = 2$.

on the runtime of the synchronization procedure. The test scenario has the native components of each of two connectors exchange a single message. Three network configurations are tested, sampling the space of network latencies that can be expected: (a) for 'localhost', the connectors share a host, (b) for 'localnet', the connectors are reachable via a local area ethernet connection, and (c) for 'internet', the two hosts are reachable by a network channel over the internet whose round-trip time is estimated by `ping` to be 15 milliseconds. This plot features values for input and output variables over orders of magnitude, and so features logarithmic x and y axes with axis ticks denoting successive multiples of two.

Figure 11.15 plots the runtime as a function of message size for each of our three network configurations. In all three cases, there was no significant difference in response to varying the message size until the size exceeded some threshold. For these runs, the majority of runtime is a consequence of waiting for control messages, and manipulating control data structures, which are not affected by message length. The three network configurations have vastly different baselines, which follows from the vastly different times taken for their consensus algorithm to exchange messages over the network link. However, as messages approached their maximum allowed size, runtimes increased such that they trended toward a linear relationship with message length, suggesting that eventually, the cost of reading, writing, sending and receiving message contents predominates the runtime.

Figure 11.16 shows the results of Figure 11.15 again, but with runtimes scaled to a proportion of the runtimes for the 1-byte message runs of their respective network configuration. In this plot, it is clearer to see the threshold beyond which the increasing length of exchanged messages makes a significant difference. Sessions communicating over lower-latency transports have lower overhead overall, and thus, can be seen to experience a significant slowdown for even moderately sized messages.

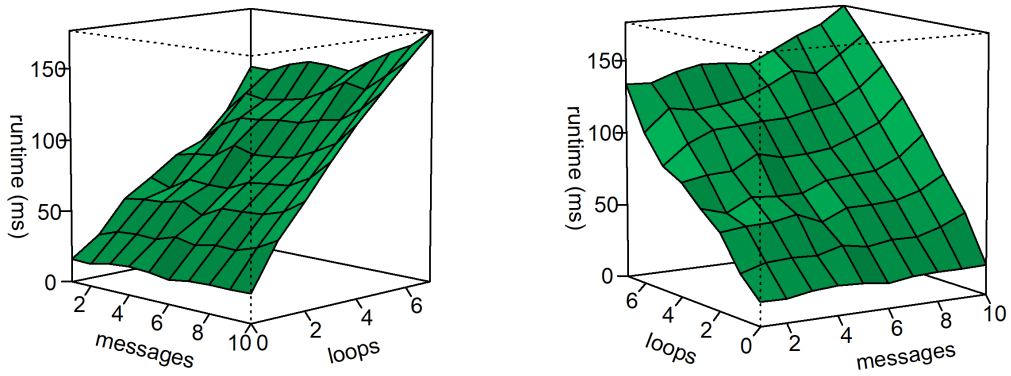


Figure 11.14: Runtime of `connector_sync` as a function of the session's configuration, plotted against combinations of parameters: (a) the number of messages exchanged in parallel from leader to follower, and (b) the number of times each message is looped from follower via the leader back to the follower. Note the logarithmic *x* and *y* axes.

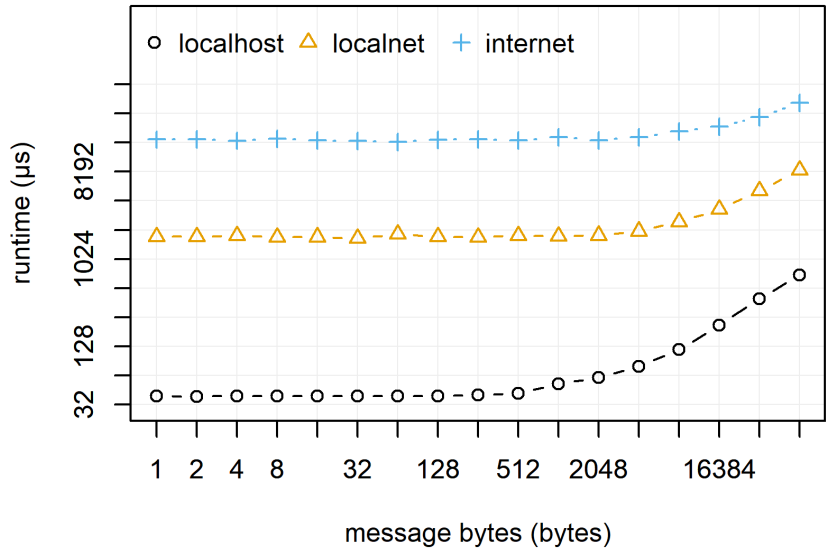


Figure 11.15: Runtime of `connector_sync` of the 'leader' in a two-connector system where the leader sends the follower a single message each synchronous round.

In anticipation of frequent moves and replications of messages, the reference implementation stores payload data on the heap, and moves them through components and ports by reference. Our final experiment shows that protocol components managed within the same connector can exchange large messages without incurring cost dependent on the message length. Figure 11.17 shows runtime measurements for a single-connector session, where the native component synchronously send and receives a same message, first sent through a loop of `forward` components hosted locally. The component graph of this session corresponds with that shown in Figure 11.10 previously, but note that, this time, protocol components are instantiated with protocol `forward`. The plotted runtime measurements are distinguished by the size of the message sent and received. Observe that while message size and the number of forward components each contribute to the runtime cost, they do so independently; successive exchanges of the same message between locally-managed protocol components don't incur the cost of moving the message contents repeatedly.

Note: Transport Strategy

The connector runtime must exchange control messages with peers over a network as part of its various control algorithms. Realizing conceptual message passing concretely requires making many design decisions pertaining to the specifics. In such cases, it is often not clear a priori which choices are superior to others, bringing rise to several permissible strategies. Here,

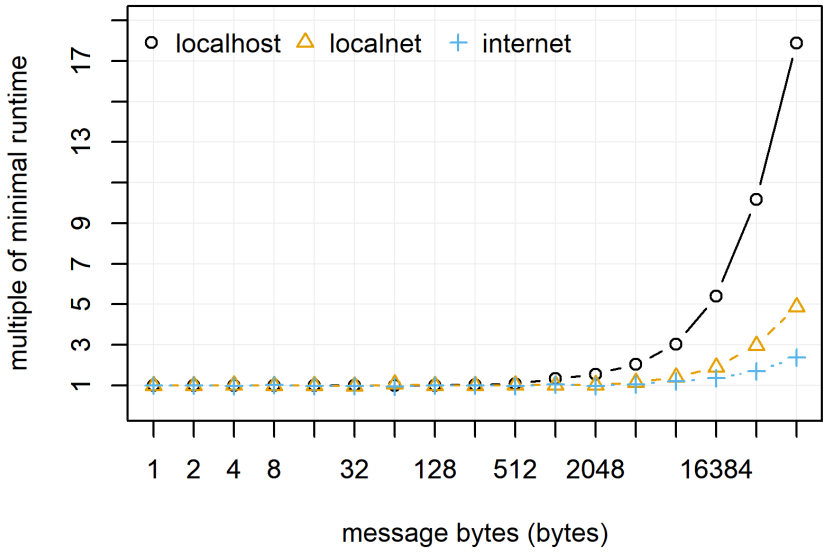


Figure 11.16: Runtimes of Figure 11.15, for each network configuration, plotted as a multiple of the respective configuration's runtime for the case of sending a 1-byte message. Note that the *x* axis is logarithmic, and the *y* axis is linear.

we focus on three strategies for the translation of outgoing control messages into TCP byte sequences for transport over the network.

1. **nagle**
   Control message structures are serialized into the TCP stream, with each incremental sub-structure written eagerly with a system call. The TCP socket is configured to use the default socket configuration, which uses Nagle's algorithm for combining small outgoing TCP segments into larger ones.

2. **eager**
   Control message structures are serialized into the TCP stream, with each incremental sub-structure written eagerly with a system call, each of which eagerly sends the data 'on the wire' as a TCP segment.

3. **buffered**
   Control message structures are serialized into a byte buffer incrementally. Once the serialization is complete, the entire buffer's contents are sent 'on the wire' as a single TCP segment with one system call.

Figure 11.18 examines a test scenario where two hosts connected by a set of 16 channels in either direction, backed by transport channels with approximately 1 millisecond of latency, exchange *N* messages in parallel. The figure distinguishes mean runtimes for runs employing

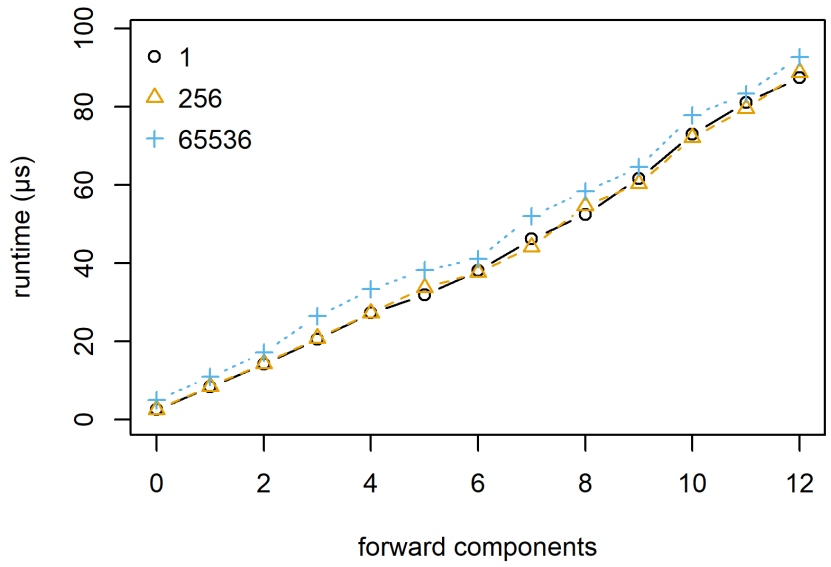


Figure 11.17: Runtime of `connector_sync` of the only connector in a session where the native component sends itself a message, after it is forwarded through a ring of `forward` protocol components. Runtimes are shown as a function of the number of components in the ring, with runs distinguished by the size of the message sent (in bytes).

each of the three strategies. 'greedy' can be seen to scale poorly with the number of control messages exchanged, as more messages result in very many TCP segments, resulting in immense transport-layer overhead, often requiring entire TCP headers to transport a single byte of control data. 'nagle' scales well with the number of control messages, but results in the operating system waiting extended periods of time to aggregate control data, resulting in consistent, yet low throughput. 'buffered' results in fewer, larger TCP segments sent very quickly, but requires an extra step of moving bytes out of the incrementally-populated byte buffer.

Overall, it is clear that 'buffered' by far outperforms the other two strategies. Clearly the overhead of the added buffering step is worthwhile. As such, it is used in the final version of the connector runtime.

11.1.3 Summary

The experiments in this section clarify the relationship between a session configuration and the performance that users can expect of their programs during communication.

As is the case in general, the most significant contributor to slowdown in the latency of the network transport itself. This is certainly true for sessions spanning the internet between nations and continents, but is also likely to hold even on a moderately-sized local area network with the typical millisecond or two of latency. The effects of such latency are offset by the execu-

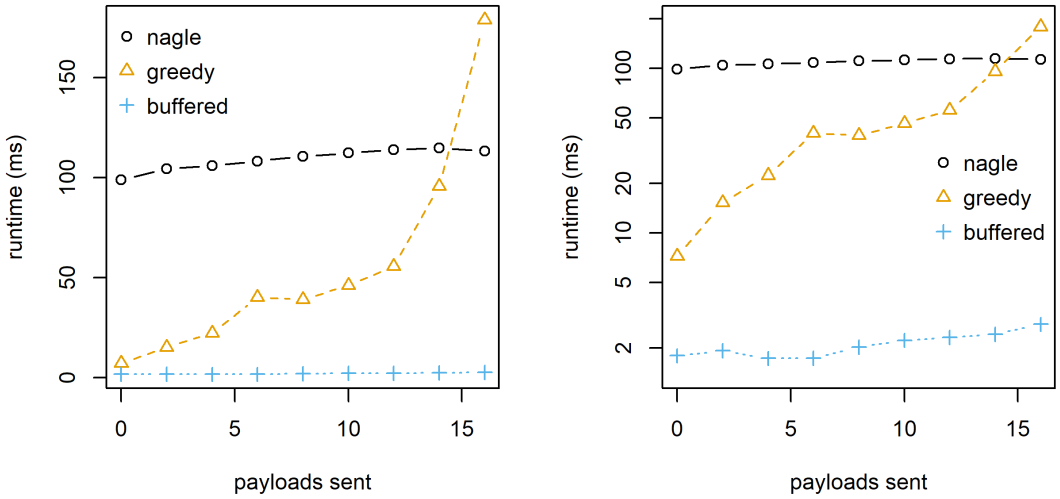


Figure 11.18: Runtime of `connector_sync` as a function of the number of speculative messages sent, results for each of the three strategies are shown for rendering outbound control messages are tested. The right figure mirrors the left, but with a logarithmic y-axis. **Note** that all results but for those of 'buffered' were made using a variant of the runtime implementation, and as such, are not reproducible using the publicly available source code.

tion of tasks in parallel, which is generally achievable for actions within the same synchronous round, inhibited primarily by the presence of causal dependencies between the actions. For example, two applications in a local area network can expect to exchange five small independent messages in less time than it takes to bounce a single small message from one host to the other and back again. Sections 12.2.6 and 12.1.1 describe the potential to relax the notion 'synchrony', enabling the exploration of multiple rounds' actions in parallel, with the goal of increasing throughput.

There is generally no significant difference in the cost of transporting messages less than a thousand bytes in length. The cost of transporting longer messages only becomes significant once they contain several dozen kilobytes of data, and then in those cases, the cost of repeated message movement are limited to the cases in which messages traverse the boundaries between connectors, over the network. Section 12.2.2 describes future work to improve the performance of speculative messaging over the network. Section 12.2.4 describes future performance optimizations of the handling of incoming network messages. Section 12.2.8 describes changes to the connector API that will afford more efficient expression of behavior on behalf of the user, affording clear and efficient deduplication of API calls.

Generally, the cost of synchronization of a session scales with the complexity of its protocol configuration in the sense that 'you pay for what you use'. Complex protocols involve a large set of ports, a large set of components, protocol components defined to perform length computations, and protocol components that perform extensive speculation of the possible behaviors. The most complex sessions involve many of these factors at the same time, such that the complexities are compounded, E.g, a session with multiple components, each performing significant speculation. Sections12.2.5 and 12.2.3 describe changes to PDL and the connector implementation, respectively, which promise to optimize the runtime performance of protocol components. Section 12.1.5 describes a scheme for relaxing restrictions on the number of messages transmissible through ports per synchronous round, in some cases reducing the number of ports needed to express a given abstract protocol. Nevertheless, in the cases of realistic protocols, the performance impact of the session's protocol is overshadowed by network delays by several orders of magnitude (a matter of microseconds versus milliseconds).

11.2 Effects of Session Optimization

Chapter 8 introduced *session optimizations* as a distributed procedure for mutating the initial configuration of the distributed connector runtime during the setup phase. In most cases, the utility of this procedure its its ability to replace the session's internals such that the resulting session is more efficiently executable than it would have been otherwise, but without altering its behavioral specification.

This section exemplifies the sorts of session optimizations available to connectors, giving a glimpse into their efficacy by selecting simple, concrete example cases, and comparing the runtimes of the session with and without the optimization. It is beyond the scope of this work to perform an exhaustive exploration of the kinds of optimizations possible, or to make subjective recommendations of which optimizations are best suited to any particular use case. Rather, the goal is to simply illustrate the space of possible optimizations such that it may be more systematically explored in future work.

For the sake of brevity in the sections to follow, if not otherwise specified, user messages exchanged at ports have a length of 1000 bytes.

11.2.1 Collapsing Idempotent Component Chains

As is the case for the `sync` component in Reo, `sync` components in PDL are unique, in that they have a symmetry with the synchronous channels through which components exchange messages. Despite this apparent redundancy, sync components are very useful when programming in PDL, primarily of a way to join an existing input and output port together. Perhaps the most essential session optimization is the collapse of a sync component (with its input and output ports) such that the two incident channels are unified. Without this optimization, the runtime dutifully manages the sync component as any other, incurring all the associated overhead.

Figure 11.19 shows runtime measurements of a simple connector, in which a session is configured to redirect an application's messages back again. A reasonable implementation of this setup is given below in the C programming language; concretely, the native's messages are routed back to itself via a single sync component:

```
#include <time.h>
#include "reowolf.h"
int main(int argc, char** argv) {
    Arc_ProtocolDescription * pd = protocol_description_parse("", 0);
    Connector * c = connector_new(pd);
    PortId p0p, p0g, p1p, p1g;
    connector_add_port_pair(c, &p0p, &p0g);
    connector_add_port_pair(c, &p1p, &p1g);
    connector_add_component(c, "sync", 4, (PortId[]){p0g, p1p}, 2);
    connector_connect(c, -1);
    /* communication goes here */
    return 0;
}
```

Figure 11.19 shows the mean duration of a communication round for a session with this configuration; without any optimization the sync component is kept as-is, incurring a cost of 12.15μs whenever a message of 1000 bytes is sent. However, if the `connect` procedure includes an optimization pass to remove the sync component, this is shaved down to 2.62μs. This 4.63-fold speedup may make a considerable difference to the performance of the user's program if it features in a critical loop. consecutive occurrences of sync components may arise in practice, particularly as the result of other session transformations; the figure includes runtimes for longer chains of consecutive components, showing how, with each collapse, runtime is incrementally reduced.

This same optimization can be applied to any chain of components that are 'idempotent', in that a chain of length 1 has the same behavior as a chain of length > 1. For another example, consider the class of components that filter incoming messages, before synchronously forwarding them to their output.

11.2.2 Localizing Components

Findings in Section 11.1.2 make clear the costs of causal dependencies between components managed by distributed hosts, as it requires extra time for the control information to propagate over the network than within shared memory. This next optimization shows an exploitation

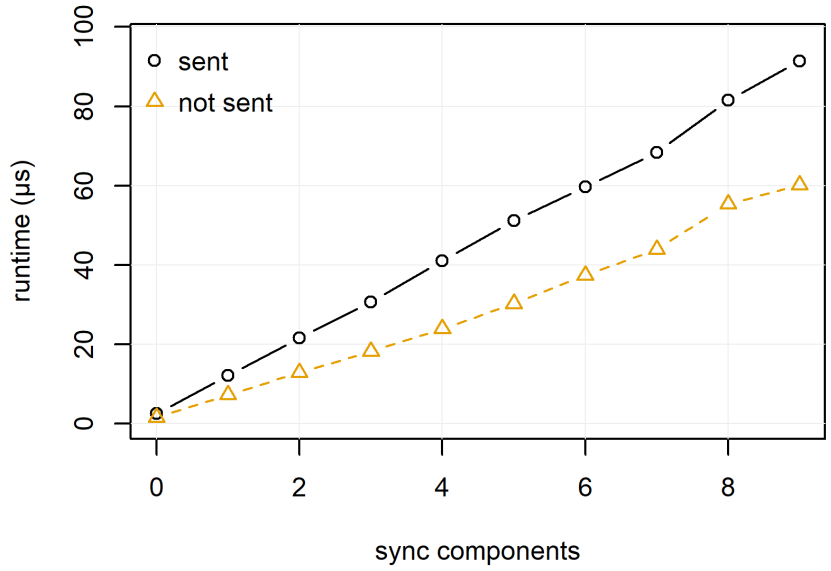


Figure 11.19: Runtime of `connector_sync` in a single-connector session comprised only of the native component connected as part of a ring of `sync` components. Runtimes are distinguished for rounds in which a kilobyte message of user data is sent around the ring or not. Results are plotted against the number of sync components in the ring.

of an optimization opportunity that is only possible at session time: two hosts join over the network, and the result of the transformation is for one to adopt the work of the other, such that the session as a whole is more efficient. Concretely, our example involves two applications (equivalently, native components) A_0 and A_1 with connectors C_0 and C_1 respectively, which interface via a pair of network channels, one in either direction. A_1 retains ownership of its two ports, while C_0 passes them to an instantiated `sync` component. The result is a session where A_1 receives their own messages by bouncing them off C_0. Figure 11.20 visualizes this session before and after this optimization as a pair of component graphs. The implementations of the setups of A_0 and A_1 are given below:

```
Arc_ProtocolDescription * pd = protocol_description_parse("", 0);
Connector * c = connector_new_with_id(1);
PortId ports[2];
EndpointPolarity ep = EndpointPolarity_Active;
connector_add_net_port(c, &ports[0], addr, Polarity_Putter, ep);
connector_add_net_port(c, &ports[1], addr, Polarity_Getter, ep);
connector_connect(c, -1);
```

```
Arc_ProtocolDescription * pd = protocol_description_parse("", 1);
Connector * c = connector_new_with_id(0);
PortId ports[2];
EndpointPolarity ep = EndpointPolarity_Passive;
connector_add_net_port(c, &ports[0], addr, Polarity_Getter, ep);
connector_add_net_port(c, &ports[1], addr, Polarity_Putter, ep);
connector_add_component(c, "sync", 4, ports, 2);
connector_connect(c, -1);
```

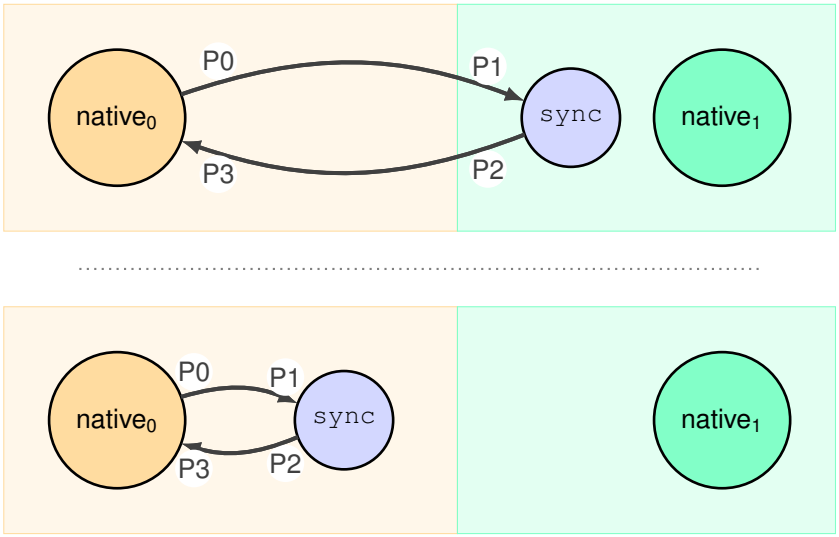
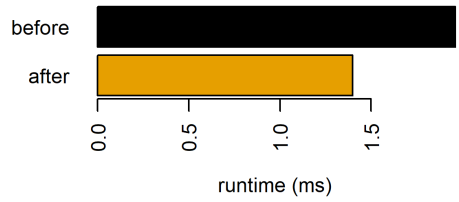


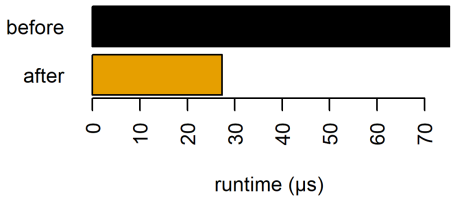
Figure 11.20: Before (above) and after (below) applying an optimization to localize the communications between the native component of connector 0 with a `sync` component initialized by connector 1.

Without any further optimizations, this implementation is perfectly functional, but suffers slowdown as a result of the connectors not taking advantage of the available information; C_1 still awaits the outcome of C_0 performing its work every round, despite the fact that this work can be more cheaply performed locally, saving both connectors the effort.

It is out of the scope of this project to define a robust implementation of a session optimization that recognizes and optimizes for this particular case; the distributed control algorithm for



(a) Connectors span a local area network with round-trip time estimated by `ping` to be 1 millisecond.



(b) Connectors reside on the same host, and use the localhost network for connectivity.

Figure 11.21: Runtime of `connector_sync` of a session with two connectors, where one 'bounces' a user message off the other each round. Each subfigure shows the effect of a session optimization which localizes the `sync` component which bounces the messages. Subfigures show different network environments; note the different scales of their y-axes.

session optimization is in place, but is not smart enough to perform the transformation. Instead, we provide a proof-of-concept by testing the effects of a handcrafted session optimization which moves the `sync` component from C_0 to C_1. The exact implementation is provided in the following snippet, taken from the (temporary) implementation of this hardcoded transformation, written in the Rust programming language:

```
// move the only protocol ccomponent from info0 to info1
let (cid, component) = info0.proto_components.drain().next().unwrap();
info1.proto_components.insert(cid, component);

// update info1's ports to route locally, rather than via the network
for port_info in info1.port_info.map.values_mut() {
    port_info.route = Route::LocalComponent;
}
```

The results of this optimization are shown in Figure 11.21. The resulting speedup varied in response to the latency in the network; in the case with higher latency, the result is a greater decrease in runtime but a lower *proportional* decrease in runtime (i.e. lower speedup); the un-optimized runtimes are likely benefitting from *latency hiding*, as the speculative control messages of round N traverse the network partially in parallel with the consensus control messages of round $N - 1$. In either case, the effect of the optimization is a significant speedup. Also note that the resulting network becomes eligible for the 'sync collapse' optimization described in the previous section, enabling further speedup.

11.2.3 Composite to Primitive

There are many ways of expressing the same observable behaviors as protocols in PDL, some of which are terser than others. It is fruitful to recognize these behavioral equivalence classes, such that instances of them in the session can be swapped out for whichever member of the class has the most desirable runtime characteristics. For example, we consider examples of canonical Reo protocols expressed in Section 10.2.2 as composite components, i.e., protocols defined in terms of simpler protocols.

The following primitive component definitions of `sequencer3` and `xrouter` are equivalent their respective composite components definitions. However, they both result in the creation of far fewer components and ports:

```
primitive sequencer3(out a, out b, out c) {
    int i = 0;
    while(true) synchronous {
        out to = a;
        if      (i==1) to = b;
        else if(i==2) to = c;
        if(fires(to)) {
            put(to, create(0));
            i = (i + 1)%3;
        }
    }
}
```

```
primitive xrouter(in a, out b, out c) {
    while(true) synchronous {
        if(fires(a)) {
            if(fires(b)) put(b, get(a));
            else         put(c, get(a));
        }
    }
}
```

Figure 11.23 shows the effects of optimizing the `xrouter` in the simplest case: a single connector receives signals from the protocol component, which is used as a generator of signal messages through one of three ports in a cyclic fashion, where all channels transport messages in shared memory. The only change observable by the user is the 4.55-fold speedup. The configuration of this session is depicted in Figure 11.22.

For the case of the `xrouter` component, measurements are more complicated, as its effects on runtime performance depend on the behavior of the session, as it expresses a nondeterministic choice in how its input message is routed. Figure 11.24 shows the effects of replacing the composite definition of `xrouter` with a primitive. Figure 11.24a shows the results grouped by the native component's behavior, making it easier to see the speedup of applying the optimization in either of three example usage cases: (a) 'left' the native component always receives messages from the left port, (b) 'right', where the native component always receives messages from the right port, and (c) 'cyclic', where the native component cycles between receiving from

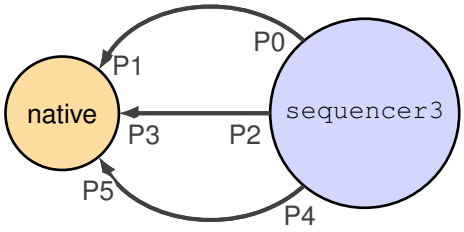


Figure 11.22: Component graph of a single-connector session where a native component receives token messages generated by a `sequencer3` protocol component through three logical channels.

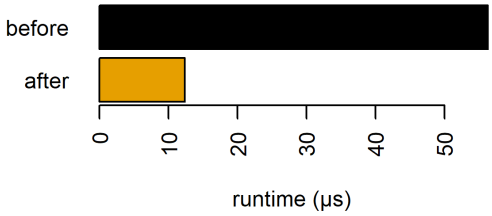
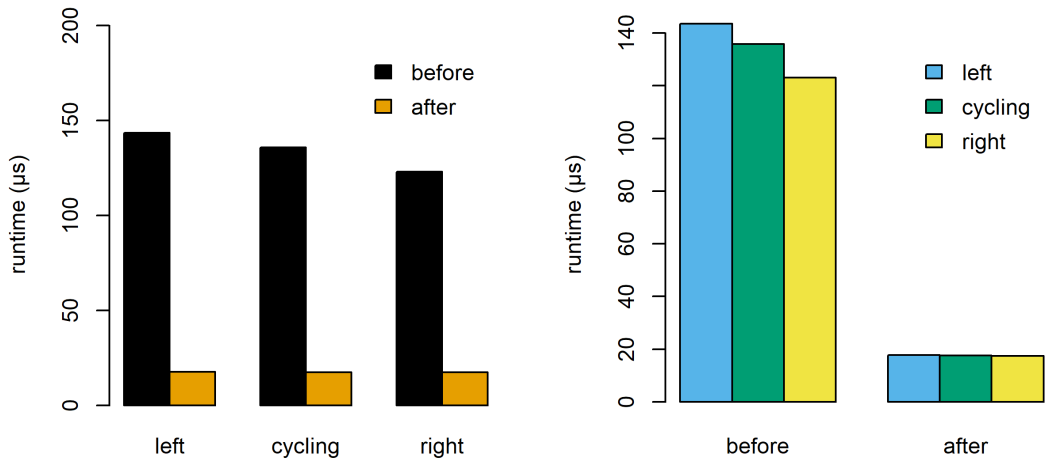


Figure 11.23: Runtime of `connector_sync` in a single-connector session where the native component receives signal messages from a `sequencer3` protocol component through a sequence of ports in circular, alternating fashion, one per round. Bars distinguish mean runtime with and without a session optimization to swap-in a primitive definition for `sequencer3`.

the left and right ports. Speedup is shown to range from 7.04 to 8.06. Figure 11.24b shows the same results grouped by whether or not the optimization has been applied; in this plot, its clear to see that aside from consistent speedup, the primitive definition results in a runtime less dependent on the native component's behavior. When defined as a composite component, the resulting session is comprised of a large network of interlinked primitives, where the speculations of one, results in the others speculating also, which comes at a cost.

11.2.4 Specialization

Nondeterministic branching allows a component to express flexibility to a particular outcome, enabling other components, or the runtime environment itself to make the choice. In cases where one component's flexibility is consistently constrained by a neighbor, the result is speculation on outcomes that will never be involved in accepted solutions. When such a case is identified during the session optimization step, its possible to replace the affected components with others that don't bother considering the doomed options. This is referred to as *specialization*, as it results in the same general-purpose protocols being distinctly specialized as a function of their environment.



(a) Grouped by native behavior (b) Grouped by protocol definition

Figure 11.24: Runtime of `connector_sync` of a session with a single connector, whose native component receives their own message each round through a `xrouter2` component. The native decides which way the message is routed by choosing which of the two available ports it receives from, either always receiving from the left port, always from the right port, or cycling between them. Runtimes are distinguished by whether the `xrouter` is provided a composite or primitive definition.

Figure 11.26a shows the results of replacing a `sync` component with a `forward` when its outputs are passed to another `forward`. This is a case where the `sync` would explore a non-deterministic choice (it either does or does not receive a message from its input) needless in the greater context of its environment. Making this replacement has the effect of a moderate speedup of 1.17. However, more importantly, this transformation facilitates other replacements whose results can compound to result in even greater speedup. Concretely, a next step can collapse the chain of `forward` components into a singleton as was seen in the example `sync` components in Section 11.2.1.

Figure 11.26b shows the more considerable speedup achieved by removing two components' non-deterministic choices, whose outcome is always irrelevant to the rest of the session's components. As depicted in Figure 11.25, a `xrouter` component routes incoming messages to one of its output ports as the result of a non-deterministically. Likewise, a `merger` component accepts up to one message through either of its input ports as the result of a non-deterministic choice, and forwards them to its output. Together, this cluster has the same interface and observable behavior as a single instance of `sync`; messages that go into the only input, synchronously come out the only output. To make the fairest case for the session's performance 'before' the optimization, the definition for `xrouter` is given the more efficient primitive definition shown previously in Section 11.2.3.

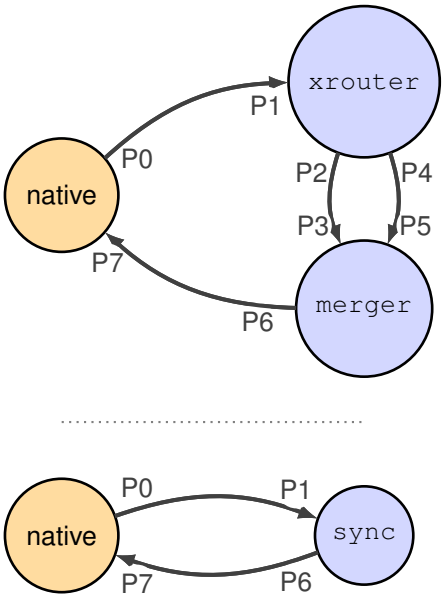
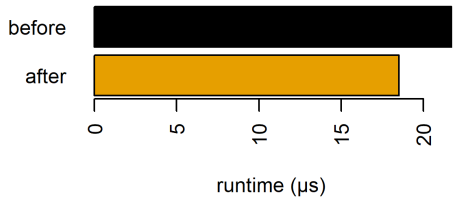
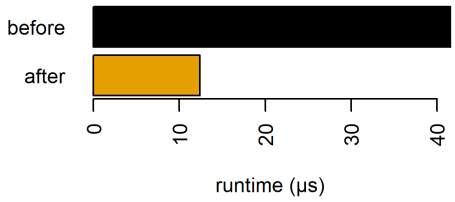


Figure 11.25: Component graphs showing the result of a single-connector session where a native component send and receives their own messages, passed through a coupled `replicator-merger` component pair, before (above) and after (below) an optimization to remove the unnecessary non-deterministic branching.

Both of these test cases are relatively unaffected by changes to the user's message size. This is expected, as the optimization removes only speculative branches which deal with aliases of messages in shared memory; as discussed in Section 11.1.2, within shared memory, duplication of user messages is done by reference, incurring a small cost independent of its message.



(a) Components `sync` and `forward`, connected in series, replaced by `forward` and `forward`



(b) Components `xrouter` and `merger` joined by two channels replaced by single `sync`

Figure 11.26: Runtime of `connector_sync` of a session with a single connector, where a cluster of components are specialized to their session context, replaced by another cluster with the same observable behavior, but facilitating more efficient execution.

11.2.5 Minimizing Network Traffic

In some cases, protocol components handle messages flowing from one connector to another. For these cases, it is more difficult to reason about which connector should manage the component's state; either way, the network will have to be traversed somewhere in the message path. This section shows examples of fruitful session transformations in cases by moving components to connectors in such a way that network traffic is reduced overall.

A very simple example which facilitates a very obvious benefit in the best-case scenario, involves a filtering component, initially configured to filter messages from connectors *A* to *B* on *B*'s side (the incoming side). By moving the component to be managed by *A* (on the outgoing side), messages filtered out are not needlessly sent over the network. This optimization is depicted in Figure 11.27. The definition of `filter` is provided below:

```
primitive filter(in i, out o) {
    while(true) synchronous() {
        msg m = get(i);
        if(m[0] == 0) put(o, m);
    }
}
```

Figure 11.28 shows the effects of applying this optimization; the speedup is greatest when there is less network latency, and larger messages, as these cases cause the act of repeated control message transmission to take up the largest proportion of the runtime; the test with the best result achieves a 1.7-fold speedup.

Just as significant is the reduction in bytes exchanged over the network overall per round, shown in Figure 11.29 to trend towards half as message sizes grow; this matches our intuition, which says that in the event the message isn't received by receiver, the sender's message need not be communicated over the network. The total number of bytes sent over the network in a

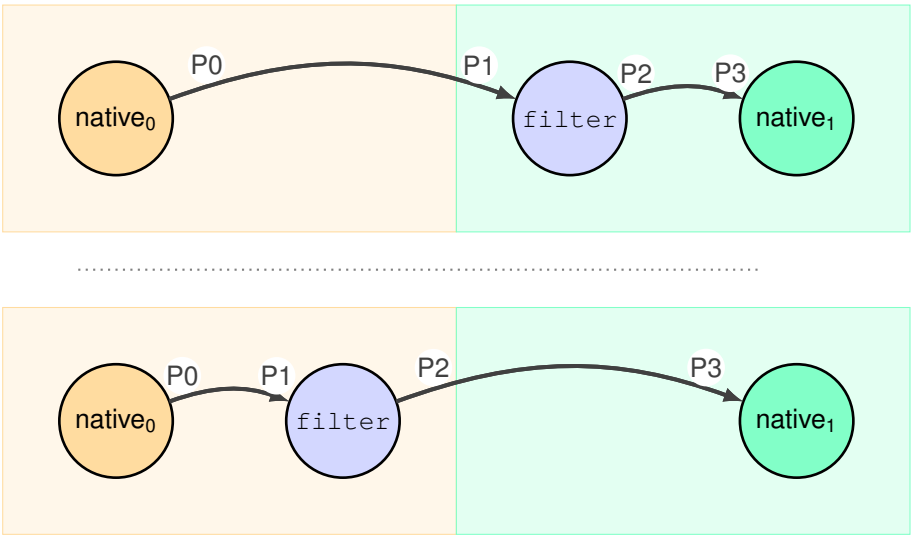


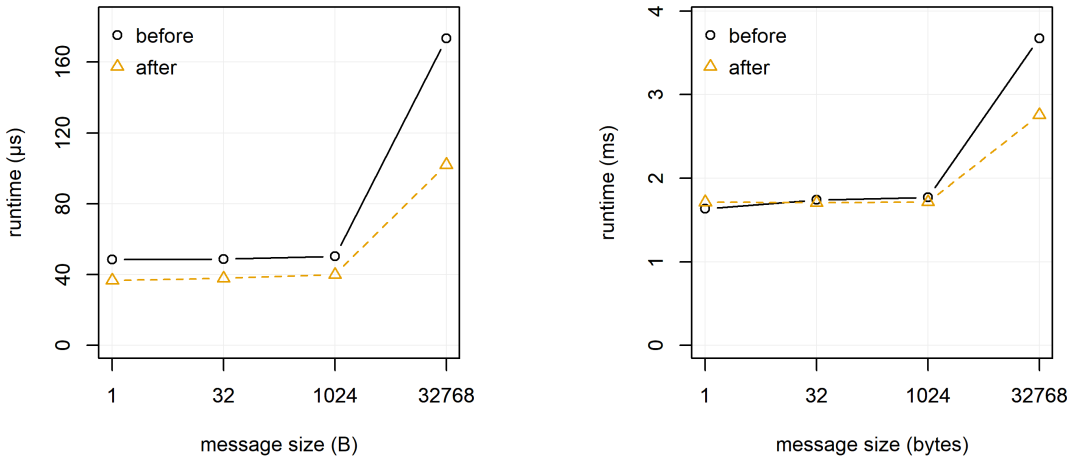
Figure 11.27: Before (above) and after (below) applying an optimization to move a `filter` component closer to the source of its incoming messages.

given round for sessions with this configuration is given by R:

$$\begin{aligned} R &= \frac{R_S + R_F}{2} \\ R_S &= 2B_S + B_A + B_P \\ R_F &= B_S + B_A + N \cdot B_P \\ \hline B_S &= 7 + 3S_S \\ B_A &= 7 + 3 * 3 \\ B_P &= 4 + 3 * S_S + B_M + B_{B_M} \\ \hline S_S &= 1 + N \end{aligned} \tag{11.1}$$

where the values of the variables can be understood as follows:

R_S	Total bytes exchanged during the round in the event the user message succeeds in passing the filter
R_F	Total bytes exchanged during the round in the event the user message fails to pass the filter
B_S	Byte-size of a 'suggestion'-type control message, sent from connector B to A as part of the consensus procedure
B_A	Byte-size of a 'announcement'-type control message, sent from connector A to B as part of the consensus procedure
B_P	Byte-size of a 'send-payload'-type control message, sent from connector A to B as part of the speculation procedure
B_M	Byte-size of the user message traveling from applications of connectors A to B
B_{B_M}	Byte-size of the variable-length encoding of B_M.
S_S	Number of speculative variable assignments in 'suggest'-type control messages
N	acts as a flag, having value 1 if the session optimization is NOT applied



(a) connectors share a localhost

(b) connectors share a local area network

Figure 11.28: Runtime of `connector_sync` of a session with two connectors, where connector *A* sends a message toward *B*, which the filtered out by a `filter` component. Plots show runtimes before and after an optimization which relocates the filter from *B* to *A*. Note the logarithmic x-axes, showing the runtime responding to choices of message length. Sub-figures distinguish measurements by the network configuration.

From the equations, it is more clear precisely how this session optimization affects network traffic, by examining both occurrences of the variable N. Once enabled, (a) no 'send-payload'-type control message is sent in the event the user message fails to pass the filter, and (b) a minor, incidental contribution is the reduction in the number of speculative variable assignments included in 'suggestion'-type control messages from B to A as a side-effect of one channel being moved from B to A along with the filter component. It also becomes clear how to generalize total runtime or network latency as a function of P, the proportion of rounds in which the user message passes through the filter. The effect of this optimization is really to achieve $\frac{1}{1-P}$ speedup as the size of the message trends toward infinity, with our test runs performed with $P = \frac{1}{2}$.

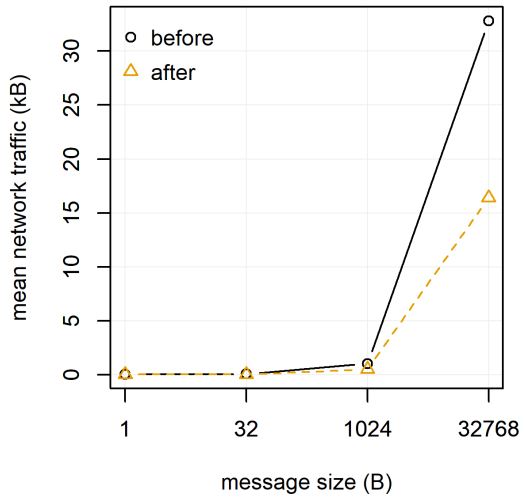


Figure 11.29: Mean network traffic per synchronous round for the results shown in Figure 11.28.

Our final experimental case once again involves a session with two connectors, A and B. This time, the latter receives a message through each of the two network channels bridging the connectors. An optimization presents itself when A threads a `replicator` component onto its ports toward B, such that B always receives a pair of replicas of a single message sent by A's native component. This time, the opportunity for optimization presents itself as the possibility to move the replicator closer to its destination, such that messages are replicated in their path *after* they have traversed the network. This optimization is visualized in Figure 11.30.

We expect a result similar to that of the previous experiment, as we expect the number of overall network messages to be halved by applying the optimization. More specifically, we expect the overall runtime of the synchronous round trends towards half as a result of applying the optimization as the message size increases. Figure 11.31 shows that this is indeed the trend, but it does not converge as quickly as in the previous example. This is the result of another significant overhead not being halved this time: latency as the result of a causal dependency over the network. In sessions with this configuration, with or without the optimization, connector B doesn't reach a satisfactory solution until the user message(s) complete their transport over

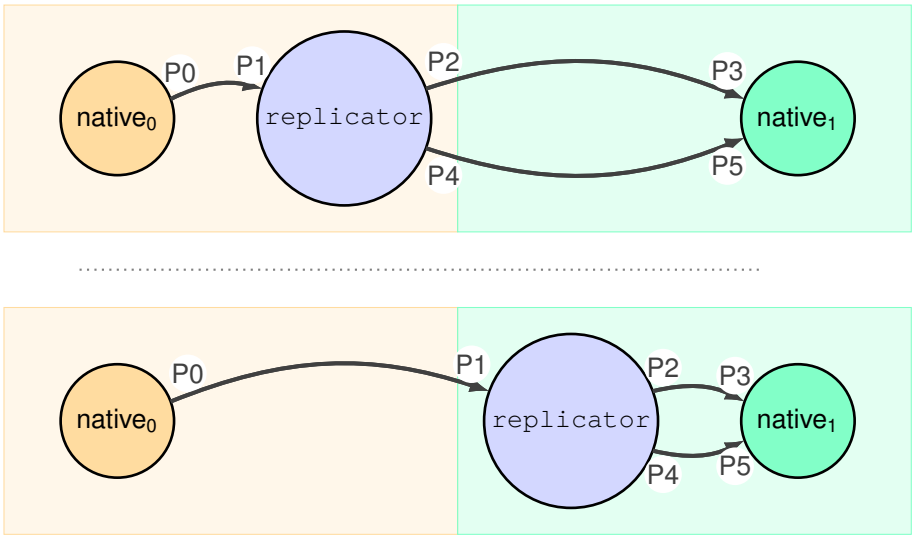
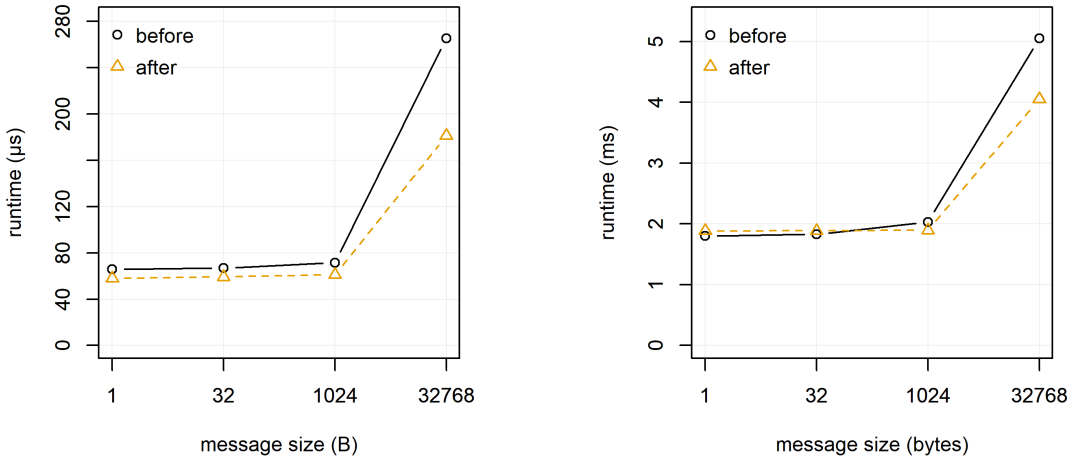


Figure 11.30: Before (above) and after (below) applying an optimization which moves the `replicator` component over the network, closer to the destination of its outgoing messages, such that there is an overall halving of messages being sent over the network.

the network; in the before-case where there are two messages, latency hiding occurs, as the time taken for each message to traverse the network overlaps.

The proportion of remaining network traffic as a result of applying the optimization also trends to half as the size of the user message increases. As this is unaffected by any latency-hiding effect, the effects are observed far sooner, with the sum of all other forms of control messaging always resulting in less than a hundred bytes of control data per synchronous round.



(a) connectors share a localhost

(b) connectors share a local area network

Figure 11.31: Runtime of `connector_sync` of a session with two connectors, where connector *B* receives two replicas of some message sent by *A*. Plots show runtimes before and after an optimization which relocates the `replicator` from *A* to *B*. Note the logarithmic x-axes, showing the runtime responding to choices of message length. Sub-figures distinguish measurements by the network configuration.

Chapter 12

Future Work

This section presents a collection promising directions for future work on PDL and the connector runtime implementation. Section 12.1 is primarily concerned with developments that make fundamental changes to PDL and the observable behaviors of sessions. Section 12.2 is primarily concerned with refining the internals of the connector runtime to improve its robustness, flexibility, and performance.

12.1 PDL Developments

This section describes promising future work oriented around the refinement or expansion of PDL, changing the set of protocols that can be expressed, or changing their realization in the implementation.

12.1.1 Unbounded Non-deterministic Choice

The expression of non-deterministic choice is essential to both Reo and PDL for defining behavior flexible to the constraints of the environment. In PDL, components are able to describe branching behavior as a function of *speculative variables*, functioning as entry points for the runtime's branching speculative search. Currently PDL offers only `fires` as an expression (parameterized by a port variable) whose evaluation describes a binary speculative branch enumerating at most the values `true` and `false`. This proved to be a sensible design choice, as it allows for an intuitive means to both (a) fork the speculative execution in two, and (b) constrain the permitted value of a port variable. Currently, protocol components lack a means of doing the former, without also doing the latter. The example protocol `sync_eq` in Section 10.1.2 demonstrates how the absence of this functionality can result in cumbersome workarounds.

We conceive of a future extension to PDL to support arbitrary non-deterministic choice. As an example, consider a new intrinsic procedure `choose`, which is evaluated to an integer within a given range. In the following example, the component sends one of two distinct one-byte messages, as a function of the expression `choose(2)` evaluating to either 0 or 1 at runtime.

```
primitive foo(out p) {
    synchronous {
        msg m = create(1);
        if(choose(2)==0) m[0]=42;
        else             m[0]=100;
        put(p, m);
    }
}
```

Alternatively, `choose` is a keyword decorating a block of *N* statements, expressing that exactly one statement be chosen non-deterministically for execution at runtime.

```
primitive foo(out p) {
    synchronous {
        msg m = create(1);
        choose {
            m[0] = 42;
            m[0] = 100;
        }
        put(p, m);
    }
}
```

In future, it should be investigated how to provide the flexibility that `choose` provides, without creating an undesirable redundancy arising from the overlap between the functionalities of `fires` and `choose`. Perhaps `fires` could be eliminated altogether, with its work done by a combination of `choose` (to introduce a branch) and `assert` (to constrain a port value).

12.1.2 Unifying Primitive and Composite Components

PDL enforces a strict dichotomy between *primitive* and *composite* components; the former are able to participate in synchronous rounds by communicating, and the latter are able to create new ports and components.

The current implementation of the connector runtime only requires a weaker property: every component must perform composite-only work between synchronous blocks, and perform primitive-only work within synchronous blocks. This is because the only real limitation is the lack of support for creating and destroying components while speculation is underway, as the former modifies the structure of the *solution tree*, which is used to drive the latter.

In future, the benefits from the added flexibility of blurring the lines between these two categories of component may be found to outweigh the detriments. Syntactically, this may be as simple as removing the `primitive` and `composite` keywords altogether, and permitting these unified connectors to act like composites and primitives whenever, as long as its without and within synchronous blocks respectively.

12.1.3 Relaxing Synchrony to Atomicity

Recall how the semantics of PDL are based on the Table Model, which lays out the observable behavior of a system at runtime as a table, with ports for columns, round numbers for rows, and the cells at their intersection containing the messages exchanged at a port during a round (for more details, refer to Section 5.1). This approach can be understood as orienting the timing of message exchange for all the connectors' components to some shared clock. Aside from being easy to understand, this approach gives the power to components to control the timing of their actions, which manifests in PDL as the predictable relationship between the occurrences of the `synchronous` keyword, and the sequence of interactions that have been realized at runtime. With this approach, the session at large makes progress if and only if every component makes progress. This is a double-edged sword, as both (a) components can rely on the runtime's efforts to realize interactions preventing the component's starvation, but also (b) the session cannot progress if any component cannot progress.

Future work could explore the effects and feasibility of removing this property by weakening our current notion of synchrony. Components would no longer be in control of how their actions are grouped into the session's interactions. Taking inspiration from previous work on the Reo language, we could envision components reasoning about the *atomicity* of their actions instead, grouping them into blocks with `atomic` keywords, much as they do with `synchronous` keywords now. Behavior contained within an atomic block expresses the necessity for it to all occur as an indivisible unit, i.e., succeeding together as part of a single interaction. In a nutshell, where synchrony allows components to control in precisely which round each of their actions will be observed, atomicity allows components to group their actions into the same interactions, but have no control over which. In a nutshell, atomicity takes control away from components, and gives it to the runtime.

As an example of atomicity in action, consider a session with two components, mutually linked by three logical channels, whose ports each component agrees are called *a*, *b*, and *c*. Despite the protocols grouping their local actions differently, their session may nevertheless realize their behavior as a single interaction in which the zero-length message is sent through all three channels:

```
primitive foo(in a, in b, in c) {
    atomic { get(a); get(b) }
    atomic { get(c); }
}
```

```
primitive bar(out a, out b, out c) {
    msg m = create(0);
    atomic { put(a, m); put(b, m); put(c, m); }
}
```

Sessions built from components unable to control how their actions are synchronized may be desirable, because (a) no component has the inherent power to inhibit the session's progress at large, and (b) the connector runtime implementation may enjoy better throughput as a consequence of being freed of strict synchrony requirements.

12.1.4 Consensus Tree Reconfiguration

A limitation of the current implementation, is the inflexibility of the *consensus tree*, the overlay structure imposed atop connectors during the *connect procedure*, described in Section 6.2.3. This inflexibility has consequences for the connector, as an inflexibility of the relationships between connectors, particularly prohibiting the addition or removal of connectors once communication has begun.

Ideally, future work would investigate the addition of new, robust control algorithms that enable the reconfiguration of the consensus tree once the session has started. Such developments are conceivable, in the best case utilizing an efficient distributed algorithm exist for various manipulations of trees. In the worst case, the implementation could simply be extended with functionality that results in the distributed tear-down of the current session's tree, such that it can be newly constructed. Given this capability, new session behavior becomes possible; for example:

1. **Interleaving setup and communication**
   A significant portion of the user-facing connector functionality is already available in either the setup or communication phase. For example, applications are able to inject new protocol components into their sessions both during the setup phase, and in-between participation in synchronous rounds. The primary limitation is the inability to create new ports over the network, as this would otherwise include synchronous communications with a connector not yet included in the consensus tree. In future, this functionality would be permitted by re-configuring the consensus tree to include any newly-reachable connectors.

   In the event this new channel is created between two connectors already in the communication phase, the result may be the merging of their sessions (and consensus trees).

2. **Removing connectors without ending the session**
   Currently, sessions only continue while all connectors remain present. Permitting connectors to leave arbitrarily between synchronous rounds requires the reconfiguration of the consensus tree to 'cut out' their node without resulting in a partition.

   A clue for a means of approaching this feature is recognizing that connectors with no children (i.e., leaves in the consensus tree) are able to leave without partitioning the tree. Thus, it would suffice to introduce a control algorithm which moves a connectors leaves to become the children of its parent. While simple in theory, correctly implementing this functionality in a large distributed system robustly is no trivial task.

Taken a step further, the ability to reconfigure the *solution tree* (the super-tree of the consensus tree which has component leaves), in the middle of the synchronous round, enabling the addition and removal of components while speculation is in progress. This feature would be particularly useful if combined with that described in Section 12.1.2. This is likely a very difficult problem, and may instead be better solved by a novel approach that makes more essential, systemic changes, such that the runtime may do away with the solution tree altogether.

12.1.5 Unbounded Messaging per Round

Currently, components are able to use a port for the exchange of up to one message per interaction. This is a limitation shared by Reo, the language on which PDL is based. As a consequence of this limitation, the complexity of an interaction is bounded by the number of ports.

Future work could investigate the effects of removing this limitation, allowing each port to send or receive a sequence or set of messages per interaction. The resulting behavior would be

comparable to allowing components to create extra ports on-the-fly on a round-to-round basis. In the case of message sequences, this added functionality may not require any changes to the syntax of PDL, but simply permitting behaviour that is currently prohibited. For example, the following `foo` protocol would be interpreted as receiving a sequence of two, distinct messages:

```
primitive foo(in a) {
    synchronous { get(a); get(a); }
}
```

Supporting this functionality requires changes to the synchronization procedure, and the representation of candidate predicates. For example, rather than speculating on *whether* a port fires, they can speculate on the *number* of firings per port; perhaps predicates constraining the number of messages sent by a putter port would incrementally raise a lower bound of this number, while for getter ports, they would lower an upper bound, such that particular solutions tighten both bounds to some particular number of messages sent and received.

12.1.6 Extended PDL Backwards Compatibility

During the work which resulted in Chapter 9, the functionality of connectors was extended to support the creation of intrinsic components able to interface with a remote peer over UDP, exposed as an input/output port pair for use in the session. The result is the ability to involve peers in communication sessions, without declaring the behavior of the hosts across the UDP channel; this is similar to the way that native components allow user applications written in a language such as C to participate in communications without declaring their behavior in detail.

In future, connectors may be able to represent an larger set of low level system resources in this explicit manner. Taken to the extreme, existing network protocol stacks can be implemented in PDL such that their protocol descriptions look nearly indistinguishable from C. This reveals the first advantage to supporting this paradigm: a large body of legacy implementations are more easily ported into PDL, and usable as part of sessions for connector communications. The extension of PDL must be handled delicately, as this trend might encourage the expression of protocols at a low level, which is more difficult for humans and automated tooling to reason about. However, if done well, a larger share of the network stack can be included into the connectors themselves, making possible more extensive and subtle session optimizations, and allowing for reasoning about safety properties to extend deeper into the protocol stack.

As a concrete goal, a future work might pursue the developments necessary to make it possible to express a working DNS client in PDL.

12.2 Implementation Developments

This section describes potential future work oriented around refining and optimizing the internals of the connector runtime. These can be understood as changes that impact the user's experience in terms of runtime performance or ease of use, without making a significant change to the conceptual design of PDL.

12.2.1 Rule-based Session Transformation

Future work could develop the connectors' *session transformation* procedure, introduced in Section 8.1. Currently, the foundation for extensive transformations is in place, but lacks a robust and automatic means of reasoning about the available session information. These transformations can take inspiration from the example 'proof of concept' transformations, showcased alongside their effects on runtime performance in Section 11.2. Ideally, these ideas would be taken a step further, to the extent that user-defined protocols would be reasoned about in terms of their abstract properties.

For example, rather than identifying chains of `sync` components, and replacing them with a singleton, a generic session transformation may search for chains of any identical components identified as having 'idempotent' effects on the messages they transmit.

A scheme such as patch graph rewriting [OE20] shows promise for driving such a session transformation procedure, providing a systemic approach to pattern-matching and transforming the session's components piece-wise by reasoning about patch transformations about a connector graph.

If the runtime cost of reasoning about sessions and protocols becomes a concern in future, such work is complemented by a sheme to 'offload' reasoning and transformation work offline. We can envision a paradigm of the connector runtime 'looking up' safe session transformations from a persistent database, populated through the offline verification of canonical protocols, or verification of generic protocol properties.

12.2.2 Session-Wide Symbolic Message Passing

Described in Section 8.2.1, the runtime implementation move messages between connectors very efficiently within a single connector's shared memory. This optimization was introduced to address the most prevalent cause of message duplication: speculative branching; a component with a large set of user messages in its store, or having a large set of messages on its input ports results in significant message replication when its state is forked to explore exclusive speculative branches. However, Section 8.2.2 goes on to explain that the benefits of this approach are conferred to other situations also; cases of message replication *within* a single speculative branch are cheap also. The most obvious example is within the `replicator` protocol. Section 11.1.2 demonstrates this optimization's influence on runtime performance, including its applicability to messages within shared memory only.

Future work may investigate the benefits of extending the current system to allow for safe message replication without this limitation. For example, we can envision a scheme where every connector will reason about the uniqueness of incoming messages, aliasing an existing replica whenever possible, regardless of the source of the message. To see how the current system can be generalized, consider how the current scheme effectively introduces an (implicit) identifier to the contents of messages, realized as the pointer to the memory location where the contents are stored. When a message is replicated in memory, the replica is ultimately a reference to this memory location. The problem is simply the failure to resolve this memory location for incoming network messages, and thus, being forced to resort to copying and storing the contents redundantly.

One approach is to introduce an identifier space for payloads, which aims to minimize (ideally, to one) the set of identifiers used for any given message regardless of where it originates. A practical approach may be to rely on a hash function, and message-content storage structure per connector, where messages stored by components and transmitted over the network represent their contents symbolically: the hash of their contents. Taken to the extreme, this en-

tirely decouples the transmission of messages from the transmission of their contents; perhaps the latter could be circumstantially avoided altogether? Consider and example session, with messages of one connector, A, 'bounced' to another connector B, and back again. If B never inspects the message's contents, the message contents never need to traverse the network at all, as their hash value suffices to identify the message. Some problems remain, left as future work, such as: (a) How can one avoid the hash values of arbitrarily large message values colliding without control messages having to transport the message contents? (b) How can one make efficient use of the decoupling of messages from their contents? E.g., should connectors eagerly transport message hashes, and only request message contents from the source lazily, at the moment they realize that access to the contents is needed locally?

12.2.3 Specialized Component Storage Data Structure

Aside from the latency incurred by the time taken for messages to propagate over the network, the connector runtime's most time-consuming task is querying, modifying and reasoning about the states of speculative component structures.

In future, the storage of component structures can be further optimized, specialized for the tasks most frequently used during the synchronous round: (a) lookup by speculative predicate, and forking to further the search through the tree of speculative oracles. For more details on these processes, refer to Section 6.1.2. Here it suffices to know that the connector runtime stores and manipulates a (potentially) large set of small component data structures to represent the state of the session.

As is the case in general, developing a data structure specialized for optimal performance requires extensive experimentation and careful performance benchmarking, which is left as an exercise for future work. Here, we lay out some initial approaches whose likely suitability is based on idiomatic approaches for optimizing these kinds of use cases.

The storage for this structure is currently implemented as a hash map, with predicates acting as keys. This is appropriate for its common usages, as it guarantees that (a) component lookup is fast, given a predicate, and that (b) for every unique predicate, there is a unique component structure. However, this fails to take advantage of the structure inherent to predicates themselves. Below, two promising designs for the speculative component storage are suggested, having in common that they store the component structures themselves contiguously, and rely on a secondary data structure which annotates the components, such that they can be queried given an oracle predicate.

1. **Optimize for cache locality**
   The more terse the annotation structure, the more effectively it can be cached, resulting in fast traversal. Recall from Section 6.1.2 that oracle predicates can effectively be thought as a mapping from the domain of port variable identifiers to a ternary domain with values corresponding to $\{\mathit{fires}, \neg\mathit{fires}, \mathit{unspecified}\}$. Given a fixed sequences of port variables, this means that each component can be annotated by a predicate represented a sequence of pairs of bits (two bits are sufficient for distinguishing the three possible values). Queries of the connector storage can be encoded as bitwise operations, allowing multiple port variables' values to be checked in parallel. Copying a component structure is also cheap, as the cheapness of copying its predicate annotation is guaranteed by virtue of its bit-representation being small.

   For example, a system with port identifiers $\{a, b, c\}$ might encode the key-value storage of some component structure with predicate $\{a = \neg\mathit{fires}, b = \neg\mathit{fires}, c = \mathit{unspecified}\}$ with

the annotation bit-sequence of `101000`, where `10` and `00` encode ¬*fires* and *unspecified* respectively, and valuations of port identifiers are shown in the order *a*, *b*, *c*.

2. **Optimize for traversibility**
   In practice, there may often be very many port variables active in a session at a time, while the number of port variables meaningfully assigned (given value *fires* or ¬*fires* in the predicate) is usually far smaller. This *sparsity* calls for a structured data structure, which queries the set of matching components in the storage structure by focusing only on those meaningfully assigned by the query predicate, and ignoring the rest as much as possible. Many conceivable structures are possible, but one example is that of a ternary tree, for which internal nodes distinguish the three possible valuations for port variables (in {*fires*, ¬*fires*, *unspecified*}), and roots point to component storages associated with the predicate encoded in the path from the root of the annotation tree to the leaf node.

   For example, the storage of a component structure with predicate annotation {*a* = *fires*, *b* = *unspecified*, *c* = *unspecified*} could be represented by a single-edge path from the annotation tree's root, assigning *b* to *fires* (and leaving all other ports implicitly *unspecified*). Later, when another structure is added with predicate annotation {*a* = *fires*, *b* = *fires*, *c* = *unspecified*}, the annotation tree acquires a new leaf node, which shares the first edge encoding *a* = *fires* with the existing leaf, but reachable by the next step *b* = *fires*, where the existing leaf is reached through *b* = *unspecified*. Note how the complexity of performing these queries remains unchanged regardless of what port identifiers aside from *a* and *b* are used in the session.

12.2.4 Contiguous Incoming Messages Buffer

The current implementation of the connector runtime relies on TCP channels for its internet control messaging transport. As is often the case, the transmission of discrete messages, i.e., UDP-style datagrams, over TCP's bytestream requires the recipient to buffer and re-assemble datagrams on their end. Rather than reach read call writing precisely one datagram's contents into the buffer, many datagrams may arrive together, or they may arrive partially. Management of the partially-populated input buffer becomes complex when the receipt of such partial messages becomes interleaved as a result of managing several TCP streams.

The solution currently employed is simply to manage a unique buffer per TCP stream, such that the partially-received datagrams of one do not interfere with the storage of the partially-received datagrams of another. The biggest downside of this approach is the potentially large memory footprint of connectors, and the resulting sub-optimal use of the cache.

Assuming that a future implementation continues to use TCP as its control message transport, a future implementation may instead investigate a scheme optimized for minimizing memory footprint and memory copying by employing a scheme of *bump allocation* within a single, large byte buffer per connector. This is possible if even partially-received datagrams have a known size, achievable by datagrams carrying their size as a prefix. If done well, nearly all reads and writes of datagram bytes for the entire connector will be from the same memory region, making efficient use of the cache.

12.2.5 Further Specialize PDL for Runtime Execution

PDL was introduced as a variant of the Reo language, such that it may be specialized for the task of runtime interpretation by the connector runtime. As evidenced by its comparison to the Reo language, seen in Section 10.2, PDL protocols still adhere closely to their Reo equivalents.

A future implementation may rely on a representation of PDL further specialized for the task of runtime interpretation. Taken to the extreme, the Reo language itself can be used for reasoning (by humans or machines), while PDL becomes more terse and explicit. In this future, the relationship between Reo and PDL can be similar to that between Java and Java's bytecode language, relying on tools to convert from one form to the other as needed. The most obvious benefit of specializing PDL in this manner is making interpretation at runtime more efficient. However, there may be benefits simply in abandoning human readability as a design goal, allowing other design goals to be prioritized, perhaps resulting in the addition of powerful features that would not be considered for PDL in its current form.

12.2.6 Unbounded Speculative Depth

The connector runtime works to find satisfactory interactions one round at a time to advance the state of the session and facilitate communication without violating its configured protocol. Recall from Section 6.1.2 that this is achieved through the interplay between two complementary procedures: *speculation* and *decision*. The former is the search for the next interaction that satisfies the connector's protocol. The latter decides a some such interaction in particular, and commits to it, finalizing its effects and ending the synchronous round. As it stands, this decision is made by the *leader*, some elected connector host, at the moment it finds the first satisfactory interaction each round. As both processes begin and end within the round, the current implementation can be understood to have a *speculative depth* of one, meaning that it will consider the effects of interactions one round into the future before making its decision. Consequently, users can rely on the runtime not to decide on an interaction which will lead to a protocol violation within one round, i.e., immediately after taking effect.

The processes of speculation and decision may be further decoupled as a result of future work, such that the implementation is able to make decisions for interactions based on a speculation of the session's state multiple rounds into the future. Decisions can only be based on information about the future which is certain not to change after the fact. This may limit the ability of the runtime to rely on decisions relying on behavior in the distant future, particularly when the session contains many unpredictable components. Making the feature useful would require that even native components are able to suggest several rounds' worth of speculative behavior, with a promise that once the decision is made, the behavior in question cannot be undone or avoided. For example, in combination with the the developments explained in Section 12.1.3, components that reason only about atomicity and not synchrony guarantee that they are willing to participate in any number of interactions to which they will contribute no actions.

12.2.7 Kernel Level Implementation

Currently, the boundary between the user- and kernel- levels for an application running atop connectors is between the connector and its transport endpoints; i.e., connectors use system calls to act on the transport endpoints they manage internally.

In future work, the connector runtime could be implemented deeper in the kernel, such that the interface between user- and kernel-levels exists between the application and its connector(s). There are many potential benefits to such a change.

In complex sessions, connectors need to make extensive use of the transport layer to send very many control messages. If implemented in the kernel, there is less overhead incurred as the connector does this work, with added opportunities to make use of network resources more directly. For example, in kernel mode, it becomes practical for the connector runtime

to manage control messages at the level of IP packets, taking control over the necessary re-transmission and ordering control logic itself, using its knowledge of the session's protocol to make better-informed decisions. Taken to the extreme, the connector could make its willingness to re-transmit a speculative message a function of its estimation that the message will be necessary for the success of the interaction. In general, we might investigate any such opportunities for low-level system control to become informed by the session's protocol.

12.2.8 Asynchronous Automaton Connector API

Recall that communication sessions are composed of a network of components connected by ports. User programs are able to participate in these sessions, each represented as a native component which sends and receives messages through its ports one synchronous round at a time, as does any other component. The connector API is responsible for exposing this functionality to the user such that it can be incorporated into their application code. Whatever the specifics, the connector API must facilitate the delivery of constraints on the communication in the round to come from the user application, and deliver the results of the round back again.

Currently, the connector API is designed to be as familiar as possible to socket programmers, offering procedures for initiating the sending or receipt of a message. Alone, this is insufficient for expressing the desired inter-message relationships. As such, additional procedures are added to relate previously submitted message operations. Programmers may be familiar with this idiom, usually referred to as the *builder pattern*, whereby a large structure is incrementally built through method calls that incrementally refine some internal state until the contents are ready, whereupon they are explicitly finalized. Section B.2.2 details this part of the connector API.

In future, the connector API may be rewritten to use a new idiom entirely, oriented around the user application and the connector sharing access to some large data structure which serves as the medium of information exchange. This structure could encode an automaton representing the changes in the native component's state that (a) the user application permits, and (b) the connector runtime eventually realizes. There are several potential advantages to such an approach:

1. **Fewer calls to pass information to the connector**
   The application can prepare the automaton for the next synchronous round at its own pace, without involving the connector runtime. Access can be controlled within a single call, passing it from application to runtime at the start of `connector_sync`, and being returned again at the end. The reduction in the number of calls may greatly improve performance if the connector's functionality is provided by system calls.

2. **Fewer calls to return information from the connector**
   The information returned by the connector runtime at the end of the round can be significantly complex also. Currently, the most essential information is returned by `connector_sync`, distinguishing failure from success, but the application must invoke subsequent calls with particular parameters to reflect on any other properties of the round that are of interest. For example, if the round is successful, the application may inspect the contents of some received message with `connector_gotten_bytes`; if there was an error, the application may want to retrieve the newly-populated error string with `reowolf_error_peek`.

3. **The application can reuse the automaton**
   If the connector runtime modifies the automaton very little, the user application can rely

on its structure being retained between synchronous rounds, such that they don't need to be rebuilt repeatedly. The cases which would see the most benefit would be those in which the application describes a large automaton, encoding very complex requirements, but behaving in the same manner every round.

A rework of the connector API provides the ideal opportunity for increasing its functionality to facilitate *asynchrony*, as it is commonly defined for I/O procedures: where the control flow of the caller (the application) is not blocked while awaiting the completion of the callee's work (the connector). This functionality goes well with the developments explained in Section 12.2.7, such that the connector's work be performed by dedicated kernel-level processes.

12.2.9 Empowered User Timeout

Participating in a distributed procedure ultimately puts the process in question at the mercy of other hosts in the network. In cases where an unbounded wait for completion of some process is undesirable, the implementation allows users to provide a timeout parameter, expressing the maximal duration the user is willing to wait before they would rather regain control than complete the procedure.

In the case of `connect`, a timeout event represents a failure to finish arranging the participating hosts into a distributed system. In such a case, there is really only one sensible way of abandoning the procedure in favour of returning within a given timeout duration: abandoning the procedure altogether, and reverting to the state before connection began. This is indeed the solution the implementation uses.

However, `sync` may timeout for many reasons, not all of which have the same implication on the ability to continue the session. As the constraint satisfaction problems that users are able to describe can have unknown difficulty, the implementation allows a system for any connector to `request` a timeout, short-circuiting the decision procedure to end the speculative round when finding the solution is taking longer than their user is willing to wait. However, to preserve the consistency of the connectors' views of the outcome, this procedure itself is reliant on the cooperation of peers, preventing the connector from returning immediately. This can be thought of as a 'soft' timeout, which the connector makes a best effort to respect. Section 7.1.2 illustrates that, for cases where something more fundamental goes wrong with the session, the user cannot rely on their timeout duration being respected by `sync`; if a neighboring peer fails (intentionally or otherwise) to facilitate the completion of the timeout request, `sync` may be stuck waiting.

Future work may provide users with a finer degree of control by supplying several timeout arguments to `sync`, communicating the time the user is willing to wait before the runtime resorts to increasing drastic measures to restore control to the user: first, requesting a distributed timeout, and then at the worst case, abandoning the session. There are also more subtle intermediate solutions of arbitrary complexity; for example, the connector can attempt to reconfigure the consensus tree to 'cut out' an unresponsive neighbor, using the functionality resulting from the future work described in Section 12.1.4.

12.2.10 Identifier Reuse

To facilitate all of a session's connectors creating port and component resources in parallel, the allocation space of identifiers is partitioned over the set of connectors a priori. More specifically, each connector with id X manages the allocation of identifiers matching (X, Y) for any Y. To keep things simple, connectors chooses contiguous, increasing values for Y when creating new allocations, starting from zero. The space of identifiers is so much larger than any realistic

number of ports and components that there is no need to free components. The monotonic increase in Y is not a practical concern either, as the number of bits needed to represent their value grows logarithmically.

In future, where the runtime is required to support connectors that communicate indefinitely, and create infinitely many ports and components over the session's duration, it will be necessary to introduce a robust system for freeing these allocated identifiers. There are two complications that prevent the most obvious approaches: (1) thanks to session optimization, components or ports may be destroyed while managed by a connector other than that which allocated its identifier, and (2) the connector runtime cannot weed out copies of old identifiers from the states of its components (consider the native component as the best example of such).

Problem (1) can be overcome by introducing a distributed procedure for freeing identifiers. As there are already wave-like distributed procedures in place expected to occur at regular intervals as part of the session's standard operation (E.g., the sychronization procedure), a set of identifiers to be freed can 'piggyback' on existing control messages, such that the original allocator can free the identifier once it receives notice. This approach is simple to understand, as it keeps in place the property that, given an identifier, only one connector has the authority to allocate and free it.

At first glance, the problem (2) is more difficult, as it suggests that it will never truly be safe for a connector to free an identifier of a destroyed port for fear that its previous owner will erroneously use it in future. In practice, this problem can be solved using another existing system: explicitly-tracked port ownership: the connector runtime keeps track of which component owns a port. Currently, this is in place for two reasons (a) it escalates programming errors or malicious intent on behalf of components by preventing them from using a channel owned by another component, and (b) it is necessary for the connector runtime to keep track of which ports *did not* exchange messages during the synchronous round, preventing the acceptance of interactions in which a message is 'lost' in a channel by the putter putting, but the getter not getting. In future, the runtime can be adjusted to consider that a component attempting to use a port it does not own may not be an error, but rather simply a consequence of the port identifier in question having been since destroyed and re-allocated, such that it is owned by a different component. The only remaining unsolved problem is the case in which some component is allocated a new port with the same identifier as a previously-destroyed port they also own; when this component uses the 'old' identifier, the runtime may erroneously confuse it with the new port, resulting in unexpected behavior.

12.2.11 Dense Candidate Predicate Encoding

Section 6.1 explains how predicates over candidate solutions (i.e., candidate interactions) are the primary means by which connectors aggregate and compare information at runtime. As such, predicates are represented in most control messages; they are sent, received, serialized, deserialized, and compared frequently. Currently, the values of these variables are constrained using a value from ternary domain: true, false, and 'none', the case for which the value is not mapped by the predicate.

For many data structures, it is possible to re-organize the representation of its contents to alter the trade-offs between properties such as traversibility and redundancy. Consider the example of a set of integers; such a structure fundamentally encodes $\mathbb{Z} \to \mathbb{B}$, where $\mathbb{Z}$ and $\mathbb{B}$ are the integer and Boolean domains respectively. Here are three of the many possible representations: (1) A sequence of tuples explicitly encoding mappings with pairs $\{(z, b) \mid z \in \mathbb{Z},\ b \in \mathbb{B}\}$, (2) A sequence of integers, explicitly encoding mappings to the value *true*, and

implicitly encoding mappings to *false* by omission, or (3) A sequence of Boolean values, where each ith Boolean value b expresses the mapping $i \mapsto b$.

While currently, the implementation can be understood as using approach most similar to approach (1) for representing the speculative variable assignments of candidate predicates; concretely, candidate predicates encode a partial map of speculative variables (integer pairs) to speculative values (small integers) as a sequence of tuples, where missing mappings correspond to variables whose values are unspecified.

Future work might investigate the effects of reworking the representation of candidate predicates to something more akin to approach (3), representing the partial mapping as two sequences (in lockstep) of one bit per variable whose value is known to be in $\mathbb{B}$. The first bit distinguishes its assigned value of 0 from 1, and the second bit distinguishes a mapped value from an unmapped value (in the case of the latter, overriding the value of the first bit). Such an approach has some practical advantages; for example: (a) bitwise operations could perform reasoning on contiguous, machine-word-sized sequences of speculative values in aggregate, and (b) with each variable having a known index, querying a variable's assigned value is a constant time operation.

12.2.12 Connector Identifier Re-allocation

In the absence of a system for assuredly assigning unique identifiers to connectors, we expect to rely on an overwhelming probability for the system to guess unique identifiers by drawing them from a large integer domain using a random number generator. The larger the domain, the smaller the probability of two connectors' identifiers colliding. As a consequence, we can expect for port predicates to represent these large numbers also, as port identifiers contain connector identifiers. As smaller integers require fewer bytes when using variable-length integer encoding, it may be fruitful to employ an additional optimization pass during the end of the session's setup to replace connectors' identifiers with ones that still do not collide, but such that their values are minimal. Below, we describe two possible schemes for achieving this:

- **A traversal algorithm assigns contiguous IDs**
  As part of the setup procedure, the leader can initiate some sort of centralized traversal algorithm, passing along a token. An example of such an algorithm is *Tarry's algorithm* [Fok13]). The token itself acts as an allocator for contiguous connector identifiers by recording the identifier next available for allocation, starting from zero. This approach achieves a good result, but may introduce unwanted overhead to the setup procedure, or complicate future developments which would seek to facilitate new connectors joining the session in parallel.

- **The connector ID space is partitioned recursively over the consensus tree**
  Let connectors in the consensus tree pass to their children a recursively-partitioned subspace of the integer domain, where the root begins with the entire domain, $\mathbb{Z}$. This can be easily done by representing each sub-domain with an integer prefix (encoded as a sequence of bits). For example, where $a \cdot b$ is the concatenations of the bits of integers a and b, the root begins with $\mathbb{Z}$, keeps identifier $[0] = 0$ for itself and gives sub-domain $\{[1] \cdot z | z \in \mathbb{Z}\}$ to its only child, who keeps identifier $[1] \cdot [00] = 4$ for itself and gives $\{[1] \cdot [01] \cdot z | z \in \mathbb{Z}\}$ and $\{[1] \cdot [10] \cdot z | z \in \mathbb{Z}\}$to its two children, who keep identifiers $[1] \cdot [01] = 5$ and $[1] \cdot [10] = 6$ for themselves respectively.

  The benefit of this approach is it paints a picture of means of unifying the identifier spaces of components, ports and connectors into a single space. The downside is that it introduces an unwanted correspondence between the length of a connector's path to the

leader, and the bit-size of their identifier, i.e., for 'unbalanced' trees, the identifier space will be unevenly partitioned over the set of connectors, consequently, resulting in some identifiers have very long prefixes.

12.2.13 Robust Connector Security

Section 7.1 describes the systems in place in the connector implementation to preserve correctness and make a best effort to maximize session progress when all connectors are following the control protocol correctly.

Future work can investigate changes to the implementation to formally verify the existing safety properties, and modify the implementation to strengthen its safety and liveness properties as possible. Furthermore, future work can supercede the informal descriptions of Section 7.2 by completing a rigorous analysis of the connector implementation's vulnerability to *bad connectors*, erroneous or malicious peers participating in communication sessions.

Armed with a more thorough understanding of the strengths and weaknesses of connectors, follow-up work could investigate augmentations of PDL to facilitate connectors taking extra steps to work alongside un-trusted peers safely . For example, PDL may facilitate additional annotations to its components, communicating to the runtime an unwillingness to share the component's description with peers as part of its control algorithms. Such a feature may lay the groundwork for the implementation of components tasked with sensitive work, whose behavior should be kept local to the user's machine.

Bibliography

[Fok13] Wan Fokkink. *Distributed algorithms: an intuitive approach*. MIT Press, 2013.

[OE20] Roy Overbeek and Jörg Endrullis. Patch graph rewriting. *arXiv preprint arXiv:2003.06488*, 2020.

Appendix A

Extra Benchmarking Plots

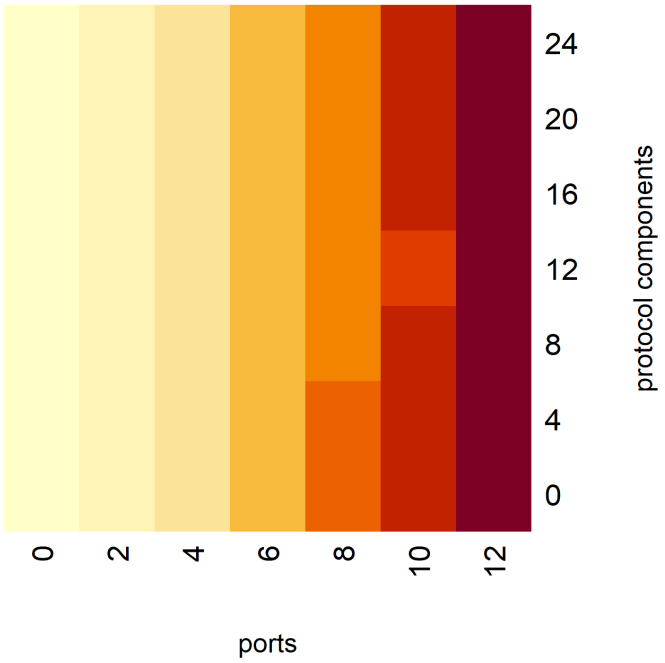


Figure A.1: Contents of Figure 11.4 rendered as a heatmap.

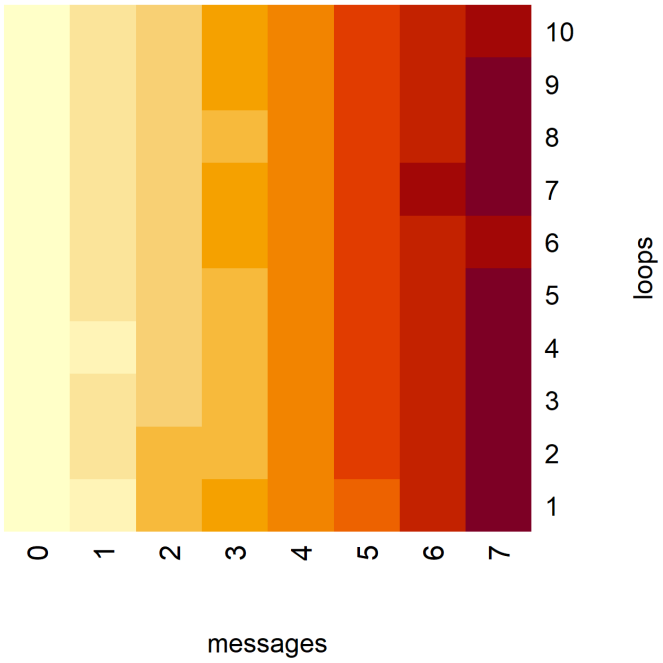


Figure A.2: Contents of Figure 11.14 rendered as a heatmap.

Appendix B

Preliminary Application Programmer Interface

This chapter serves as a historical reference for the preliminary connector API. The contents of the chapter are largely obsolete, but are retained for future reference. An up-to-date version of the connector API is given in Chapter 3.

Socket programming revolves around the creation, management, and use of sockets to exchange data with a single peer. Programmers take for granted that another application will create the counterpart socket, resulting in the manifestation of a shared, two-party communication medium at runtime, which we will call the 'socket-pair'. Applications rely on sockets as an abstraction of the rest of the socket-pair. Communication is represented abstractly, appearing to the application as the result of interactions with the socket itself. The underlying implementation is relied upon to translate these abstractions into management of underlying resources (e.g., maintaining distributed state, message-passing with IP packets, etc.). Applications are provided some control over the details of this implementation via the socket's configuration; typically, sockets can be configured with additional socket options to specialize the underlying behavior. For example, TCP supports the inclusion of the NODELAY flag to disable Nagle's algorithm for sacrificing round-trip-time on network messages to reduce messaging overhead.

Reowolf introduces *connectors* as a replacement for socket-pairs to generalize the number of connected peers (previously two) and the number of communication endpoints per application (previously one). The connector API is designed to be similar to that of sockets as much as possible. Applications represent their connector with a *connector handle* structure, and enable applications to perform abstract data exchange by reading and writing bytes to logical endpoints as before. Owing to the increased configuration space for these more complex connectors, Reowolf defines *Protocol Description Language* (PDL) for defining an application's requirements for the connector's behavior and communicating it to Reowolf during its configuration.

The API provided in this section is intended to be minimal; elaborations may be introduced to build more convenient and user-friendly APIs, using these simple procedures as their foundation.

B.1 Definitions

This section clarifies terminology to be used throughout the chapter.

1. *network address*
   This term generalizes the notion of a physical address over the internet. This generalizes over the available methods for sufficient capability to address host machines without ambiguity. For example, this may be implemented as an IPv6 address, or the tuple of (IPv4, port), where *port* is a 16-bit integer commonly used in the transport layer (e.g., TCP) for disambiguating beyond the limited capabilities of IPv4.

2. *port handle*
   Port handles are pervasive in Reowolf's API for representing logical Reowolf ports as they occur in the connector's configured PDL description. Concretely, they are non-negative integers, indicating the index of the port as parameter in the main component.

3. *port binding*
   During the setup phase of a Reowolf connector handle, the protocol's logical ports are *bound*, making clear their relationship to the application and how they are connected to peers' ports. A port given a *native* binding is exposed for data exchange to the application. *Passive* and *active* ports are exposed to the outside world, relying some application to provide a counterpart port to complete the coupling once the connector connects to its peers. Both passive- and active-bound ports are provided an address, and differ only in the means of resolving the connection: the passive port will listen passively for an incoming connection, while the active port will actively initiate an outgoing connection.

4. *message*
   The type of values being exchanged between the application and the Reowolf connector. Passed into ports with the `batch_put` procedure, and retrieved from ports with the `batch_get` procedure.

B.2 Connector Programming

Chapter 2.2 established that Reowolf connectors are inherently stateful, to reflect the stateful nature of the protocols that provide their definitions. In this section, we introduce an API for connectors such that the handles themselves are stateful also. For simplicity in this section, the API consists simply of a set of procedures; their relationship through effects on the connector's state are made explicit through text and error conditions. This approach is unlikely to be the best practice for all programming languages, requiring the design to be adapted. For example, procedures to do with communicating session-specific metadata of the configured protocol description may be exposed as the ubiquitous *builder pattern*, which creates a temporary builder object for incremental construction and applies all changes to the connector handle's state at once with a final `build` method.

Reflecting the distinct phases of a Reowolf connector outlined in Section 2.2, connector handles exist in one of four states: *uninitialized*, *connecting*, *communicating* or *closed*.

B.2.1 Setup and Configuration

Before becoming usable for communication, a connector handle must be created and set up using the `connector_configure` and `connector_connect` operations in succession.

As part of the setup phase, the application must invariably supply a protocol description, expressed in PDL. Sections to follow apply these concepts to particular programming languages, including an explanation of how textual PDL is represented in the application. Here, it suffices to

assume that the language is able to represent some textual PDL loaded into program memory by a handle that can be passed to Reowolf for inspection.

1. `connector_create`
   A new connector handle is created and returned in the fresh state. An error arises if the system has insufficient resources to support a new protocol handle.

2. `connector_configure`
   An existing protocol handle in fresh state is configured with a textual protocol description, expressed in PDL. The procedure is parameterized by (1) a connector handle, (2) the set of logical port names local to the application, and (3) a mapping from logical port occurring in the connector's configured PDL to network addresses. The connector handle moves to the connecting state.

   An error arises if the provided PDL is not well-defined. Concretely, these errors correspond with the semantics of PDL itself. For example, if a component is defined with two arguments with the same identifier. For more detailed information, see Chapter 4.

3. `port_bind_native`
   An existing connector in the connecting state expresses that the given port is accessible to the application for data exchange. The procedure is parameterized by (1) a connector handle, and (2) a port handle.

   An error arises if the connector is not in the connecting state, the port handle is invalid in the connector's configured PDL, or the port handle is has previously been bound.

4. `port_bind_passive`
   An existing connector in the connecting state expresses that a given logical port is exposed for composition with that of another application, marked to passively wait for the counterpart point to initiate the coupling at the given address. The procedure is parameterized by (1) a connector handle, (2) a port handle, and (3) the network address to which the port will bind, i.e., analogous to a TCP connection.

   An error arises if the connector is not in the connecting state, the port handle is invalid in the connector's configured PDL, or the port handle is has previously been bound.

5. `port_bind_active`
   An existing connector in the connecting state expresses that a given logical port is exposed for composition with that of another application, marked to actively initiate the coupling at the given address with a passive counterpart. The procedure is parameterized by (1) a connector handle, (2) a port handle, and (3) the network address to which the port will connect, i.e., analogous to a TCP connection.

   An error arises if the connector is not in the connecting state, the port handle is invalid in the connector's configured PDL, or the port handle is has previously been bound.

6. `connector_connect`
   An existing connector in the connecting state is connected to its peers and completes its setup phase, and entering the communicating state. The procedure is parameterized by a connector handle. The connect call returns when all ports have been connected to some peer, and likewise with any of those peers' ports.

   Errors arise if there is a problem finding peers and composing with their connectors. Reasons for this include invalid, unresolvable, or unreachable network addresses used in port bindings.

7. `connector_close`
   A connector handle is closed, freeing its underlying resources. It is an error to use an invalid protocol handle, or one in closed state. The protocol handle moves to the closed state.

B.2.2 Data Transfer and Synchronization

A protocol handle in the communicating state may be used for the exchange of data. In addition to 'communicating', these connector handles maintain an internal state of a 'staging area' for synchronous port operations. Concretely, each protocol handle maintains a set of *batches*, where each batch is a set of staged port operations, and where one batch at a time is currently *selected*. Initially, a connector handle has a singleton set of batches whose only element is empty and selected. These batches provide a means for the application to express a non-deterministic choice; The `connector_sync` call finalizes all batches and blocks until Reowolf chooses one and completes the operations within. The returned result of `connector_sync` communicates which batch was chosen back to the application.

1. `batch_put`
   Inserts a tentative put operation for a given port into the connector handle's selected batch. The procedure is parameterized by (1) a connector handle, (2) the port handle, and (3) a message. An error arises if the provided port handle is not provided a native binding. An error arises if (1) the port handle is not defined for the connector at all, (2) not defined to have direction `in` for the connector, or (3) already occurs in the selected batch.

2. `batch_get`
   Inserts a tentative put operation for a given port into the connector handle's selected batch. The procedure is parameterized by (1) a connector handle, (2) the port handle. The procedure returns a value of type message. An error arises if the provided port handle is not provided a native binding. An error arises if (1) the port handle is not defined for the connector at all, (2) not defined to have direction `in` for the connector, or (3) already occurs in the selected batch.

3. `batch_next`
   Inserts an empty batch into the connector handle's batch set and selects it. The procedure is parameterized by a connector handle.

4. `connector_sync`
   The connector handle's batches are submitted to the connector, and the call blocks until the synchronous round has been completed. Afterward, all batches are reset to their initial state, i.e., a set of batches whose only element is empty and selected. The procedure returns the index of the chosen batch as an integer. The procedure is parameterized by a connector handle. If the connector encounters an error during runtime, it is propagated to this procedure and returned to the application. For example, a protocol in inconsistent state.

B.3 Language-Specific API

This section specializes the abstract procedures of connector programming described in Section B.2 to concrete programming languages. Suggestive naming of procedures and methods preserves the mapping from the former to the latter.

B.3.1 The C API

Reowolf's C API follows the paradigms of the language by relying on a simple, procedural style. State management of resources is relegated to the application programmer, and they are expected to read and adhere to the textual documentation that accompanies the API. Connector handles are represented by file descriptors (of the `int` type) as is the convention for UDP and TCP sockets. Fallable procedures facilitate the propagation of errors to the application by use of error codes; consequently, the majority of procedures have the return type `int`.

```
#include <sys/socket.h>

/* setup */
int reowolf_connector();
int reowolf_configure(int connector, char* protocol_description);
int reowolf_bind_native(int connector, int port);
int reowolf_bind_passive(int connector, int port, sockaddr* address);
int reowolf_bind_active(int connector, int port, sockaddr* address);
int reowolf_connect();
int reowolf_close_connector(int connector);

/* communication */
int reowolf_put(int connector, int port, char* msg, int msg_len);
int reowolf_get(int connector, int port, char* msg, int msg_len);
int reowolf_close_port(int connector);
int next_batch(int connector);
int sync(int connector);
```

B.3.2 The Java API

Reowolf's Java API is more object-oriented than that of the C API, revolving around methods of the `Connector` class, with the exception of creation, which is represented by Java's canonical means for object instantiation. Fallable methods may throw exceptions for the application to handle.

```
import java.net.SocketAddress;

public class Connector {
    /* setup */
    public Connector();
    public void configure(String protocolDescription)
            throws ConfigurationException;
    public void bindNative(int portHandle)
            throws BindException;
    public void bindPassive(int portHandle, SocketAddress address)
            throws BindException;
    public void bindActive(int portHandle, SocketAddress address)
            throws BindException;
    public void connect() throws ConnectionException;
    public void close() throws CloseException;
```

```
    /* communication */
    public void put(int portHandle, byte[] message)
            throws CommunciationException;
    public byte[] get(int portHandle)
            throws CommunciationException;
    public void nextBatch();
    public void sync() throws CommunciationException;
}
```

Appendix C

Preliminary Usage & Examples

This chapter showcases a collection of example usages of connectors for network programming. Their representation is based on an outdated version of the connector API, and thus, cannot be used without modification with the final version of the connector implementation. Nevertheless, the contents of the chapter are largely retained to act as a historical reference, as many of the concepts underlying the examples are still meaningful in the current implementation. Chapter 10 gives usage and examples for the final version of the connector implementation.

C.1 Connectors

This section demonstrates the usage of connectors for coordinating message passing between a set of communicating applications. Here, all coordination is expressed by the applications themselves. To follow, Section C.2 demonstrates how applications may introduce protocol components to extend their expressive capabilities.

C.1.1 Peer Connection

The simplest use of connectors that still manages to illustrate a meaningful usage occurs when a pair of distributed peers wish to establish a shared protocol instance. In the following example, we see Amy's C application, which attempts to establish a connection with some application at a known address within ten thousand milliseconds before timing out.

```
#include <stdio.h>
#include "reowolf.h"
void amy() {
    Connector* c = connector_create();
    Port out = connector_bind(
        c,                  // which connector
        RW_OUT,             // port polarity (in/out)
        RW_CONNECT,         // method (accept/connect)
        "127.0.0.1:7000");// network ip + port

    int err = connector_connect(
        c,                  // connector
```

```
        10000);           // timeout in millisec
    if (err) printf("Error! %s\n", connector_error_peek());
    connector_destroy(c);
}
```

This case includes an example of canonical error-handling for Reowolf's C API; fallible procedures return an `int`, which gives a coarse-grained encoding for various error conditions, allowing for programmatic error handling and recovery, where zero is consistently used to encode success. In the event of a non-zero error code, `connector_error_peek` returns a pointer to a buffer containing a more detailed human-readable error description. Going forward, error handling code is omitted for brevity, along with standard `include` statements.

C.1.2 Message Passing

Next, we observe an example of two peers, Bob and Cho, using their connector to exchange a single message. As in the first example, the peers are connected by a single logical channel, and have ten thousand milliseconds to complete the connection. To connect successfully, both participants must succeed in coupling their logical ports over network channels and agree on the *polarity* of the channel, with one using it for input, and the other for output. Once connected, both applications are able to perform arbitrary local computation between synchronizations with their handle to the connector environment. Synchronization generalizes read and write calls to a synchronous, multi-party context. To allow for the larger space of expressible behavior, the application can be understood to alternate between (1) preparing a batch of port operations for the next synchronous round, and (2) synchronizing its state with the connector with the prepared port operation data.

```
void bob() {
    Connector* c = connector_create();
    Port out =
        connector_bind(c, RW_OUT, RW_CONNECT, "127.0.0.1:7000");
    connector_connect(c, 10000);

    PortOp op_out;
    op_out.port    = out;
    op_out.msg_ptr = "Hi, Cho!";
    op_out.msg_len = 8;

    connector_sync_set(c, 1, &op_out);
    connector_destroy(c);
}
void cho() {
    Connector* c = connector_create();
    Port in =
        connector_bind(c, RW_IN, RW_ACCEPT, "127.0.0.1:7000");
    connector_connect(c, 10000);

    PortOp op_in;
    op_in.port = input;
```

```
    connector_sync_set(c, 1, &op_in);
    // op_in.msg_ptr == "Hi, Cho!"
    // op_in.msg_len == 8
    connector_destroy(c);
}
```

In this case, Bob prepares a single port operation to output the message: `Hi, Cho!`. Meanwhile, Cho prepares an operation to receive some message from his peer. The `PortOp` structure facilitates fine-grained communication between the user's application and the connector runtime, and also serves as the means by which the data contents, message length and port identifiers are communicated back to the application in-place. Upon successful return, Cho's `PortOp` is mutated by `sync_*`, allowing bob to reflect on the received message at his leisure between synchronous rounds. Whether the fields of the port operation are written or read by the connector is determined by the *polarity* of the port in question.

C.1.3 Multiple Synchronous Messages

The synchronization procedure seen previously, `connector_sync_set` is named suggestively; it expresses the synchronous execution of a set of port operations. Here the behavior of connectors begins to meaningfully deviate from that of sockets. Dan and Eli express a set of message exchange operations, which either all occur together (in case of success) or none are performed at all (in case of failure).

```
void dan() {
    Connector* c = connector_create();
    Port in, out =
        connector_bind(c, RW_IN , RW_CONNECT, "127.0.0.1:7000"),
        connector_bind(c, RW_OUT, RW_CONNECT, "127.0.0.1:7001");
    connector_connect(c, 10000);
    PortOp ops[] = {
        (PortOp) {out, "Hi, Eli!", 8},
        (PortOp) {in ,  NULL     , 0}};
    connector_sync_set(c, 2, ops);
    // ops[1].msg_ptr == "Hi, Dan!"
    // ops[1].msg_len == 8
    connector_destroy(c);
}
void eli() {
    Connector* c = connector_create();
    Port in, out =
        connector_bind(c, RW_IN , RW_ACCEPT, "127.0.0.1:7001"),
        connector_bind(c, RW_OUT, RW_ACCEPT, "127.0.0.1:7000");
    connector_connect(c, 10000);
    PortOp ops[] = {
        (PortOp) {out, "Hi, Dan!", 8},
        (PortOp) {in ,  NULL     , 0}};
    connector_sync_set(c, 2, ops);
    // ops[1].msg_ptr == "Hi, Eli!"
    // ops[1].msg_len == 8
```

```
    connector_destroy(c);
}
```

The set of user-provided port operations may have any size (including zero), so long as no two describe multiple synchronous operations at the same port.

C.1.4 Expressing Alternatives

It is often useful for applications to be flexible to the needs of their peers. It is possible for an application to express the ability to offer alternative message exchange patterns to be collapsed into one concrete outcome at runtime. In this fashion, an application can be written to constrain as much or as little as possible, such that the behavior of the system is chosen to satisfy all components without the application code of one requiring complex reasoning about the states of the others.

Here we see an exchange between Flo and Gus where each expresses a set of possible outcomes for the synchronous data exchange: Flo either sends or receives a message from Gus, but not both and not neither; Gus sends no message, but he might receive one from Flo.

```
void flo() {
    Connector* c = connector_create();
    Port in, out =
        connector_bind(c, RW_IN , RW_CONNECT, "127.0.0.1:7000"),
        connector_bind(c, RW_OUT, RW_CONNECT, "127.0.0.1:7001");
    connector_connect(c, 10000);
    PortOp ops[] = {
        (PortOp) {in ,  NULL     , 0},
        (PortOp) {out, "Hi, Gus!", 8}};
    size_t *bitsets[] = {
        (size_t[]) {0b01},  // { ops[0] }
        (size_t[]) {0b10}}; // { ops[1] }
    int idx = connector_sync_subsets(c, 2, ops, 2, bitsets);
    // idx == 0
    connector_destroy(c);
}
void gus() {
    Connector* c = connector_create();
    Port in, out =
        connector_bind(c, RW_IN , RW_ACCEPT, "127.0.0.1:7001"),
        connector_bind(c, RW_OUT, RW_ACCEPT, "127.0.0.1:7000");
    connector_connect(c, 10000);
    PortOp ops[] = {
        (PortOp) {in, NULL, 0}};
    size_t *bitsets[] = {
        (size_t[]) {0b1},  // { ops[0] }
        (size_t[]) {0b0}}; // {        }
    int idx = connector_sync_subsets(c, 1, ops, 2, bitsets);
    // idx == 0
    connector_destroy(c);
}
```

The `connector_sync_subsets` procedure is shown as a generalization of that of `connector_sync_set` seen previously. Where the latter expresses a set of port operations, all of which are to be performed, the former expresses a *list of sets*, such that exactly one set is selected by the connector and performed; the application is able to determine which set was chosen by inspecting the returned `int`, the index of the set in the list.

Anticipating the frequency of use cases for which multiple port operations will be shared between sets, the C API represents the each set as a symbolic subset of a single given set. Concretely, the `connector_sync_subsets` accepts (1) reference to a set of *n* port operations, and (2) a list of *m bit sets* encoding a bit sequence of length *n*, in which a set *i*th least-significant bit encodes the presence of the *i*th port operation is present in the set. The individual effects of port operations are communicated to the application as before, by modification of the fields of the appropriate `PortOp` structures in-place. The procedure also guarantees that all but the port operations in the chosen bitset will remain unmodified.

Observe that in the context of the particular composition of Flo and Gus, the resulting behavior is entirely deterministic: Flo sends a message to Gus. No other behavior is possible, because it would violate either Flo's or Gus's constraints on the coordination. However, this would not be the case for all compositions. If two instances of Flo were initialized and connected to one another, the resulting system would exhibit non-determinism; whether the first Flo sends to the second or vice-versa would be unspecified, and therefore ultimately be determined by the whims of the connector implementation.

C.1.5 Multi-Party Synchronization

The relationship between the number of peers and the number of message channels is entirely arbitrary. In this manner, it takes no stretch of the imagination to see how connectors facilitate synchronous multi-party communication that differs only in how the ports are bound, and not in changing the program's communication logic.

The relationship between the number of peers and the number of message channels is entirely arbitrary. In this manner, it takes no stretch of the imagination to see how connectors facilitate synchronous multi-party communication that differs only in how the ports are bound, and not in changing the program's communication logic.

In the following example, we see a function which, when executed by any number of applications in parallel with pre-initialized connectors, coordinated multi-party communication occurs. Concretely, this function imposes what we will call the *secret Santa* protocol on the system for one communication round; the set of peers exchange the responsibility of buying themselves a Christmas gift with an anonymous stranger such that everyone gives and receives one gift.

```
void secret_santa(
    Connector* c,
    Port in, Port out,
    char * name, size_t namelen
) {
    PortOp ops[] = {
        (PortOp) {in , name, namelen},
        (PortOp) {out, NULL, 0      }};
    connector_connect(c, 10000);
    int err = connector_sync_set(c, 2, ops);
    assert(err == 0); // assert success
    printf("I will buy a gift for '%.*s'\n",
```

```
            ops[1].msg_len, ops[1].msg_ptr);
}
```

It is a property of all applications that successfully complete this procedure that the protcols of all participating peers (including their own) have been preserved.

C.1.6 Multiple Synchronous Rounds

Thus far, all examples have demonstrated the possible behaviors of a connected system up to the end of a synchronous round. Here, an example program is given where it is necessary for the system to experience a number of communication rounds such that the application may perform local computation in response to information exchanged previously. Below we observe an illustrative example of how the application may interleave tasks of (1) local computation, (2) reading and writing message data from previous or for future synchronous rounds, and (3) participating in the next synchronous round with prepared data.

```
void hal() {
    Connector* c = connector_create();
    Port in, out =
        connector_bind(c, RW_IN , RW_CONNECT, "127.0.0.1:7000"),
        connector_bind(c, RW_OUT, RW_CONNECT, "127.0.0.1:7001");
    connector_connect(c, 10000);
    PortOp ops[] = {
        (PortOp) {in ,  NULL                , 0 },
        (PortOp) {out, "My name is Hal!"    , 15},
        (PortOp) {out, "What is your name?", 18}};
    size_t *bitsets[] = {
        (size_t[]) {0b100},  // { ops[2]          }
        (size_t[]) {0b011}}; // { ops[0], ops[1] }
    while(1) {
        int idx = connector_sync_subsets(c, 3, ops, 2, bitsets);
        if (idx == 1) break;
    }
    size_t namelen = ops[0].msg_len;
    ops[1].msg_len = 24;
    if (namelen == 3 && strncmp("Hal", ops[0].msg_ptr, 3) == 0) {
        ops[1].msg_buf = "Hey, that's my name too!";
    } else if (namelen > 300) {
        ops[1].msg_buf = "Wow, that's a long name!";
    } else {
        ops[1].msg_buf = alloca(namelen + 18);
        ops[1].msg_len = namelen + 18;
        strncpy(ops[1].msg_ptr, ops[0].msg_ptr, namelen);
        strncpy(ops[1].msg_ptr + namelen, " is a lovely name!", 18);
    }
    connector_sync_set(&ops[1]);
    connector_destroy(c);
}
```

Hal's example is intended to illustrate how the application treats the connector as an abstraction for the rest of the network, and interacts with it in discrete synchronization events. All

other local computation work may involve arbitrary manipulation of local variables as always. In concept and in actuality, this separation results in C programs in which the coordination logic is more readily identifiable and declaratively expressed for the connector to realize.

C.2 Protocol Descriptions

In the protocol description language defined Chapter 4, the notion of connectors is strongly related to that of *protocols*, declarative descriptions of constraints on the behavior of a communicating system. Accordingly, the reference implementation exposes the *protocol description*, a data structure that can be used in conjunction with connectors at runtime to coordinate message passing. In the previous section, it was seen that C applications are able to exchange messages by creating and synchronizing with a connector at runtime through an API that affords them some ability to constrain the behavior of the system. In this fashion, connectors alone make it possible to write C applications that implement non-trivial coordination logic. However, protocol descriptions are introduced as they allow the expression of fine-grained coordination logic in PDL, a language that connectors are designed to understand. This simple property has important consequences:

1. Connectors are able to perform informed optimization, and
2. PDL is able to express relationships between synchronized messages, which relies on controlled cooperation with the connector runtime.

C.2.1 Message Equality

A surprising subspace of interesting communication protocols require that messages be related by equality. An intuitive example of such a protocol enforces equality on the contents of two provided messages:

```
primitive equal(in a, in b) {
    synchronous { // within one synchronous round
        assert(0 == msg_cmp(get(a), get(b));
    }
    // protocol ends. behavior unconstrained
}
```

However, relating messages by equality describes relationships that are even more fundamental still; synchronously forwarding a message expresses the imposition of an equality relation between the message received as input, and the message sent as output:

```
primitive forward(in i, in o) {
    synchronous {
        put(o, get(i));
    }
}
```

C.2.2 Expressing Alternatives

As is the case for the API of the connector as it is exposed to applications, protocol components are able to express alternative branches. Where the former is ‘flat’, expressed only as the

single selection between a set of port operations, protocol components are able to branch as an arbitrary function of the data they observe, including the activity of ports and the messages they receive. As an example, the `exclusive_route` component forwards a message from its input port to either one of its two output ports.

```
primitive exclusive_route(in i, out a, out c) {
    synchronous {
        msg m = get(i);
        if(fires(a)) put(a, m);
        else         put(b, m);
    }
}
```

C.2.3 Connecting Protocol Components

Previous sections introduced protocol descriptions and connectors separately. This section explains how these two ideas may be used in combination to supplement C programs with coordination logic.

Creating Protocol Components

The reference implementation exposes the `Protocol` structure, as well as a means for (fallibly) parsing a given buffer containing a textual description in PDL. While in the connected state, connectors facilitate the application instantiating components whose behavior is defined within a given protocol description, on the given ports.

```
void ivy() {
    Connector* c = connector_create();
    Port ports[2] = {
        connector_bind(c, RW_IN , RW_CONNECT, "127.0.0.1:7000"),
        connector_bind(c, RW_OUT, RW_CONNECT, "127.0.0.1:7001")};
    connector_connect(c, 10000);
    const char* pdl = "primitive forward(in i, out o) {"
                      "    synchronous {               "
                      "        put(o, get(o))          "
                      "    }                           "
                      "}                               ";
    Protocol* protocol = protocol_parse(pdl);
    connector_new_component(c,          // connector
                            protocol,   // protocol
                            "forward",  // component entrypoint
                            2,          // port array length
                            ports);     // port array pointer
    connector_sync_sets(c, 0, NULL);
    connector_destroy(c);
}
```

Protocol components created in this fashion act autonomously henceforth, progressing their states synchronously with the application's subsequent synchronous rounds. Upon instantiation, components claim ownership of their given ports, removing them from the port set of their

creator. For example, if Ivy's invocation of `connector_sync_all` included port operations with ports `port[0]` or `port[1]` would be erroneous, returning a non-zero error code.

Creating Internal Channels

As is the case in PDL, applications are able to create an input-output port pair by creating new channels between synchronous rounds. This is necessary for creating 'internal' communication such that a component may divide sub-tasks to interconnected sub-components. Consider the example of Jay's application, which connects to the network via a single input port. Jay wishes to `process` incoming messages, but cannot handle messages with a byte-length shorter than three. Rather than expressing this as part of his application, Jay defines a *forward_min_len_3* protocol component:

```
primitive forward_min_len_3(in i, out o) {
    while(true) synchronous {
        msg m = get(i);
        if (m.length >= 3) put(o, m);
    }
}
```

By working on messages that have traveled through `forward_non_empty`, the application is safe in relying on the properties it will preserve:

```
void process(size_t msg_len, char * msg_ptr) { /* omitted */ }
void jay() {
    Protocol* protocol = protocol_parse(pdl);
    Connector* c = connector_create();
    Port net_in =
        connector_bind(c, RW_IN , RW_CONNECT, "127.0.0.1:7000");
    connector_connect(c, 10000);
    PortPair mem = connector_new_channel(c);
    connector_new_component(
        c, protocol, "forward_min_len_3",
        2, (Port[]) {net_in, mem.out});
    PortOp op = (PortOp) {mem.in, NULL, 0};
    while(1) {
        connector_sync_set(c, 1, &op);
        process(op.msg_len, op.msg_ptr);
    }
}
```

This approach is strictly more flexible to Jay and his networked peers than if the constraint was expressed in Jay's application. The difference is that the protocol component is able to *constrain* the behavior of the system during the synchronous round, rather than *reacting* to it after the fact. Consider that Kim's application delegates the task of choosing between two messages, `hi` and `hello` to send through the other end of Jay's network channel. Without knowing Jay's protocol, Kim cannot know that Jay prohibits the transmission of `hi`, but accepts that of `hello`.

```
primitive forward_either(in a, in b, out o) {
    while(true) synchronous {
```

```
        if (fires(a)) put(o, get(a));
        else          put(o, get(b));
    }
}
```

The behavior of Kim's non-deterministic protocol is *composed* with that of Jay, resulting in a protocol whose behavior is the conjunction of the two each synchronous round: the message `hello` sent from Kim to Jay.