Whip: higher-order contracts for modern services

Whip: Higher-Order Contracts for Modern Services

LUCAS WAYE, Harvard University, USA
STEPHEN CHONG, Harvard University, USA
CHRISTOS DIMOULAS, Harvard University, USA

Modern service-oriented applications forgo semantically rich protocols and middleware when composing services. Instead, they embrace the loosely-coupled development and deployment of services that communicate via simple network protocols. Even though these applications do expose interfaces that are *higher-order* in spirit, the simplicity of the network protocols forces them to rely on brittle low-level encodings. To bridge the apparent semantic gap, programmers introduce ad-hoc and error-prone defensive code. Inspired by Design by Contract, we choose a different route to bridge this gap. We introduce Whip, a contract system for modern services. Whip (i) provides programmers with a *higher-order contract language* tailored to the needs of modern services; and (ii) monitors services at run time to detect services that do not live up to their advertised interfaces. Contract monitoring is local to a service. Services are treated as black boxes, allowing heterogeneous implementation languages without modification to services' code. Thus, Whip does not disturb the loosely coupled nature of modern services.

CCS Concepts: • **Information systems → Service discovery and interfaces**; • **Software and its engineering → Domain specific languages**; Interface definition languages;

Additional Key Words and Phrases: contracts, services, microservices

ACM Reference Format:
Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. Whip: Higher-Order Contracts for Modern Services. *Proc. ACM Program. Lang.* 1, ICFP, Article 36 (September 2017), 28 pages.
https://doi.org/10.1145/3110280

1 INTRODUCTION

The documentation of popular services is rife with descriptions of non-trivial properties. For instance, the documentation of the Thrift API of the popular note-taking service Evernote[1] states that the listLinkedNotebooks operation returns (among other data) a noteStoreURL, the endpoint of a service that implements the NoteStore interface, together with shareKey, an authentication token for that service. This is not a trivial property of the listLinkedNotebooks operation as it describes how another server, denoted by noteStoreURL, behaves, and how a client should interact with that service. That is, the server at noteStoreURL implements the NoteStore interface and in order to use it, a client needs to use shareKey. Indeed, this is a *higher-order* property.

Thrift and other simple wire protocols with a few simple types cannot capture such a property, let alone enforce it. As a result, the documentation describes it only informally. It is up to the developers to add code to check whether the property holds and to figure out the problem when the property does not hold.

[1] https://dev.evernote.com/doc/

2475-1421/2017/9-ART36
https://doi.org/10.1145/3110280

Despite this complication, the reliance on lightweight protocols comes with benefits. In fact, Evernote's API is just an instance of a trend in software development, which we refer to as modern service-orientation. Modern services, dubbed *microservices*, opt out of complex shared message protocols, and encourage the use of different implementation languages and the independent update and re-deployment of services. Earlier service-oriented architectures compose services using sophisticated interfaces via middleware, such as CORBA [Object Management Group 2012] or Enterprise Service Buses, but they impose complex message protocols on developers and large software shops have found that it quickly becomes a productivity bottleneck [Fowler and Lewis 2014]. Modern service-orientation makes development and deployment faster and has been employed at software companies—including Netflix, Google, Amazon, and Twitter—to construct massive and widely-used products [Fowler and Lewis 2014; Shoup 2015]. The success of modern services is succinctly summarized with the slogan "smart endpoints and dumb pipes" [Fowler and Lewis 2014].

However, as we hint at with the Evernote example, implementation errors and incorrect compositions of components are more likely, as the simple message protocols—the "dumb pipes"—make it easy to misuse a component's interface or fail to implement it correctly. For example, in Evernote's API, a noteStoreURL may be a syntactically valid URL but the endpoint denoted by it may implement a different interface than expected (e.g., possibly it provides some other Evernote service).

The inability of low-level specifications to express and check such properties leads developers to inject brittle defensive checks in their code. Misplaced or incorrect checks complicate the debugging of services. This is exacerbated when services pass unchecked (and possibly incorrect) data from messages they receive from other services. When a service eventually discovers a problem, the source of the invalid data may be multiple message hops away. To address this problem, we present Whip, a software contract system that bridges the semantic gap between the low-level interfaces of modern services and the high-level application-specific properties services need.

Inspired by Design by Contract [Meyer 1988, 1991, 1992], Whip associates each service with a *service contract*: a precise and enforceable specification of its expectations of and promises to other components. Whip service contracts embed predicates written in a full-fledged programming language in a *domain-specific contract language* tailored to the needs of modern services. Whip checks these service contracts when components run. Thus Whip service contracts make it easy for programmers to state and enforce precise conditions on the correct use of a service, and they eliminate the need for defensive code. Moreover, Whip facilitates the debugging of modern service-oriented applications, including legacy services by providing *correct blame assignment*: blame information pinpoints accurately the services whose code is the source of the bug (i.e., behavior that deviates from the service contract).

As demonstrated with Evernote's API, modern services are higher-order in nature and so should be the Whip service contracts that describe their interfaces. To that end, Whip's contract language is *higher-order* and, for instance, can express that noteStoreURL refers to a service that adheres to the NoteStore service contract, and that using noteStoreURL precludes usage of a particular authentication key. Even though contracts for higher-order functions [Findler and Felleisen 2002] have been extensively studied over the last fifteen years, adapting these results for modern services is not straight-forward. Modern services exchange data serialized as streams of bits rather than closures or objects. Thus, the Whip contract language gives to programmers specialized linguistic tools to associate serialized data with a higher-order entity of a particular interface. For instance, for the listLinkedNotebooks operation a Whip service contract can express that (i) some bits from a reply message from the operation correspond to a pointer to the code portion of a closure, i.e., noteStoreURL; (ii) some other bits from the message correspond to the closure's environment, i.e., shareKey; and moreover that (iii) the closure returns a list of maximum n notes when given a non-negative integer n (amongst other arguments).

Beyond a specialized contract language, to be useful in practice, Whip must meet a demanding set of requirements, derived from the high degree of autonomy and independence that service owners have in the implementation and deployment of their services [Fowler and Lewis 2014]:

- Whip must operate under *partial deployment*, since there may be some service owners that choose not to use Whip.
- Whip must be *transparent*: it must make no changes to communication patterns between services, so that services unaware of Whip continue to operate.
- Whip must be *language agnostic*. Services in the same application may be implemented in many different programming languages. Indeed, the source code of services may not be available to modify or even read, since application programmers wire together (possibly third-party) remote services. Thus, in contrast to a contract system for a programming language, Whip cannot depend on the runtime of a component's language to enforce contracts.
- Whip should accommodate the *loose coupling of modern services* and be *extensible with wire protocols*. The simple wire protocols used by modern services (e.g., Thrift, SOAP, JSON, and Google Protocol Buffers) enable loose coupling, but evolve over time and new ones are designed frequently.

Whip meets all of these requirements. Whip can be deployed on a service-by-service basis and is backwards compatible with non-Whip services. Whip is language agnostic, taking a black-box approach to contract checking by inspecting only the messages that services send and receive. Whip is also designed to be modular with wire protocols, allowing it to be extended to support additional wire protocols. Whip already supports popular interface technologies such as Thrift, WSDL, and REST. In other words, Whip is designed to monitor whether services adhere to their advertised interfaces in a world of heterogeneous but interoperating services where some services use Whip while other services do not use it.

Due to the above design requirements and the domain specific nature of its contract language, Whip is a complex and subtle system distinct from other contract systems. As any other contract system though, Whip aims to provide programmers with accurate information upon contract violations to cut down the debugging space and speed up fixes. Indeed, Whip *provably assigns correct blame* [Dimoulas et al. 2011]. To establish this key metatheoretical property of Whip and capture formally Whip's unique setting, we describe its behavior with a custom model, *WM* (Whip Model).

In the remainder of the paper, we first describe Whip's contract language (Section 2). Then, we provide a complete but high-level description of the runtime of Whip (Section 3). Section 4 introduces *WM* to make the informal description of the previous two sections precise. In Section 5, we use *WM* to show that Whip assigns correct blame. We have implemented Whip and used it to harden the interfaces of a variety of off-the-shelf services (Section 6), and evaluated the performance impact of Whip (Section 7).

2 THE WHIP CONTRACT LANGUAGE

We present Whip's contract language by demonstrating how it can specify precise properties of the Evernote API we discussed briefly in the previous section.

Evernote provides cloud-based storage of notes, organized into notebooks. Each user's notes and notebooks are stored in a distributed database called a *note store*. Client-side services implement tools for users to manage their notes and notebooks. Moreover, Evernote allows users to share notebooks. Thus, a note store contains the notebooks a user has created, a list of *shared notebooks* she has shared with other users, and a list of *linked notebooks* that other users have shared with her.

Accessing a shared notebook may require contacting a different note store than the one the user contacts for her own notes. Figure 1 depicts some of the steps that a client must take to access a

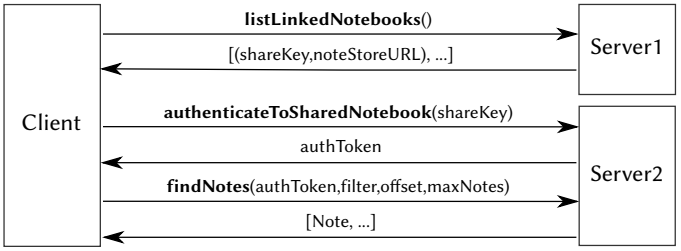

Fig. 1. Evernote: access to a shared notebook

note from a notebook shared with the client's user by another user.[2] Each box represents a service: Client is the client-side service that interacts with Evernote services; Server1 is the Evernote service that implements the user's note store; and Server2 is the note store where a shared notebook resides. Arrows indicate requests (left to right, annotated with operation and arguments) and replies (right to left, annotated with returned values).

To access a shared notebook, the client retrieves a list of linked notebooks from its note store (operation listLinkedNotebooks), and uses the information from the list to contact Server2 and authenticate to the shared notebook (authenticateToSharedNotebook). The client authenticates by presenting the particular shareKey for Server2. The client can then access the shared notebook, by, for example, calling the findNotes operation to search for particular notes, passing as one of the arguments the authToken returned by authenticateToSharedNotebook.

```
1  service NoteStore {
2    NoteList findNotes(1: string authToken,
3                       2: NoteFilter filter,
4                       3: i32 offset,
5                       4: i32 maxNotes)
6     throws (1: Errors.EDAMUserException userException),
7    ...
8
9    list<Types.LinkedNotebook> listLinkedNotebooks()
10    throws (1: Errors.EDAMUserException userException),
11   ...
12
13   AuthenticationResult authenticateToSharedNotebook(
14                       1: string shareKey)
15    throws (1: Errors.EDAMUserException userException),
16   ...
17 }
```

Fig. 2. The NoteStore Thrift API

Figure 2 displays the portion of Evernote's Thrift API that corresponds to operations that play a part in this work flow. For each operation the Thrift API describes the data types of arguments and results together with the data types of any exceptions the operation may throw.

We focus on two first-order and two higher-order properties in this work flow that are necessary to access notes from a user's shared notebooks but are beyond the capabilities of Thrift's interface description language and are stated only informally in the documentation.

First-order Properties. In contrast to its Thrift specification, the findNotes operation does not accept any 32-bit integer as its offset argument. The offset is the smallest numeric index of the notes included in the result of the operation. Thus, the documentation of the API explains, an offset has to be a non-negative integer. This is a *first-order function* property that a contract system for a programming language can capture with a pre-condition predicate.

[2] We present a simplified version of the API for clarity. Irrelevant arguments and return values of the operations we consider are elided.

The second property is a *dependent first-order function* property; the findNotes operation returns a list with length at most maxNotes (another argument to the function). Thus, it corresponds to a post-condition that states a property of the result of a function call in relation to the arguments of the call.

Higher-order Properties. The two higher-order properties of the Evernote API are the ones we mention in Section 1. First, the operation listLinkedNotebooks returns a list of pairs (noteStoreURL, shareKey) where noteStoreURL refers to a service that implements the interface of a note store. In terms of a programming language, this property can be expressed with a higher-order function contract that ascribes a contract for the services pointed to by the result of listLinkedNotebooks.

```
1  service NoteStore {
2    findNotes(authToken,filter,offset,maxNotes)
3    @requires « offset >= 0 »
4    @ensures  « length(result) <=  maxNotes »
5    ...
6
7    listLinkedNotebooks()
8    @foreach (noteStoreUrl, shareKey) in « result »
9      identifies NoteStore at « noteStoreUrl »
10     with index « shareKey »
11   ...
12
13   authenticateToSharedNotebook(shareKey)
14   @where index is « shareKey »
15   @ensures « type(result) != EDAMUserException »
16   ...
17 }
```

Fig. 3. The NoteStore contract

Second, the shareKey that is bundled up with each noteStoreURL in the result of listLinkedNotebooks has to be used as an argument for a successful call to authenticateToSharedNotebook on that service. This is a common pattern in the world of microservices due to the lack of proper abstractions such as closures and objects. Since programmers cannot properly encapsulate the environment of a piece of code, they manually follow call protocols and explicitly pass around relevant pieces of the environment of a service's operation when invoking the operation.

Whip's contract language can capture all these properties. The Whip contract language does not focus on syntactic specifications such as the data types of arguments and results of service operations (which, as Figure 2 demonstrates, interface description languages such as Thrift's already handle). Its features are tailored to the service contracts that Whip aims to express and enforce.

Figure 3 shows part of the service contract for a note store service expressed in Whip's Interface Description Language (IDL). The keyword **service** defines a service contract that describes the interface of a service, and gives a name to the contract, in this case NoteStore. For each operation the service provides, the service contract contains an *operation contract*, that is a signature for the operation followed by tags that state properties about the operation's arguments and result. For example, the NoteStore service contract includes operation contracts for findNotes, listLinkedNotebooks, and authenticateToSharedNotebook.

First-order Operation Contracts. The operation contract for findNotes (lines 2–4) expresses the two first-order properties from above: offset is a non-negative integer [3] and the length of the returned list is at most maxNotes. The tags **@requires** and **@ensures** define, respectively, a pre-condition and a post-condition, specified as Python code, i.e., code between « and » in the contract is Python. Code in pre- and post-conditions can refer to the operation's arguments (e.g., offset and maxNotes in the snippet above). Post-conditions also have access to a special variable **result** that is bound

[3]The Thrift API of Evernote already states this argument is a 32-bit integer. Thus, the Whip contract does not repeat this syntactic requirement.

to the result of an operation call. In the snippet, **@requires** checks that offset is non-negative and **@ensures** specifies that the length of the **result** list is less than or equal to maxNotes.

Higher-order Operation Contracts. Recall that the two higher-order properties of interest state that listLinkedNotebooks returns pairs (noteStoreURL, shareKey) where (i) noteStoreURL refers to a note store, and (ii) shareKey can be used to successfully call authenticateToSharedNotebook on that note store. Consider a single such pair (u, k). To express the properties, the Whip NoteStore contract must be able to express not only that u refers to a NoteStore service (say, Server2), but also that the operation authenticateToSharedNotebook on Server2 expects k as its argument.

To capture that u implements the NoteStore contract where k can be used to successfully call authenticateToSharedNotebook, we introduce *indexed service contracts*, a new kind of contract that handles this idiom of modern services. We treat NoteStore as a *family of service contracts*, indexed by a value.[4] Thus, the *indexed service contract* NoteStore$\langle k \rangle$ is an appropriate service contract for the service that u refers to.[5] The same service may, of course, satisfy other indexed service contracts from the same family, such as NoteStore$\langle k' \rangle$, where k' is a different shareKey.

Returning to the IDL, lines 9–10 present an example of a higher-order operation contract. The operation contract specifies that the result of listLinkedNotebooks identifies multiple NoteStore services: for each pair in the returned list, noteStoreURL refers to a service that implements service contract NoteStore$\langle$shareKey$\rangle$. Line 14 presents a *use* of the higher-order operation contract. This line indicates that authenticateToSharedNotebook is an operation of a service that implements NoteStore$\langle$shareKey$\rangle$, where shareKey is the operation's argument.

3 THE WHIP RUNTIME, INFORMALLY

In this section we informally describe how the Whip contracts of Section 2 are enforced by a *Whip-enhanced service*. Section 3.1 describes the high-level design and deployment of an enhanced service. Section 3.2 explains how the enhanced service uses its internal state to enforce Whip contracts, and Section 3.3 describes the enhanced service's behavior in the event of a contract failure. Finally, Section 3.4 describes how Whip leverages a special enhanced wire protocol when both services involved in an operation are enhanced.

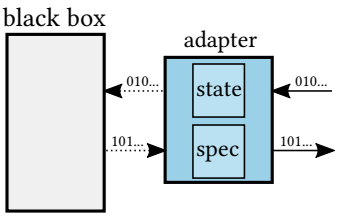

Fig. 4. Diagram of a Whip adapter

Although Whip targets distributed applications, we emphasize that we focus on functional correctness of service composition via higher-order contracts, and not on reliability or fault tolerance of distributed systems. Indeed, modern services are often chosen for organizational concerns such as loose coupling and scalability, rather than for reliability. Existing techniques to enhance the reliability of distributed systems are compatible with and orthogonal to Whip. That said, Whip assumes a communication layer, such as TCP, that can authenticate endpoints and does not corrupt messages. Whip does not rely on the order of message delivery, nor that message delivery is guaranteed. Indeed, Whip assumes a minimum structure about messages. It assumes that an enhanced service can isolate from a message (i) whether the message is a reply or a request; (ii) a unique identifier that matches a request with its reply message; (iii) the service originator of the message; and (iv) the bit string payload of the message. The unique identifiers that match request and replies can be typically based on ephemeral TCP reply port numbers. For asynchronous

[4]Indeed, every Whip contract is implicitly a family of contracts; if no index value is explicitly provided, a special default index value is used.

[5]We use "service contract" to refer to both service contract families and indexed service contracts when this is clear from the context.

messages, clients already need to associate replies with requests, and so the messages already contain identifiers that Whip can use.[6] Section 4 gives a formal account of messages, their identifiers, and how Whip processes them.

3.1 The Whip Adapter

A *Whip-enhanced service* is a service deployed with an adapter. Figure 4 depicts a deployment of a Whip-enhanced service. The Whip network adapter intercepts all messages between the service and its peers. The network adapter uses its internal state to check messages against their corresponding contract. The adapter runs in its own OS process and intercepts raw TCP data to and from the service and is responsible for checking contracts. Whip treats all services as black boxes and does not require code modifications nor does it change a black-box service's view of interaction with other services. If a contract check fails, Whip logs the details of the contract and message involved, including *blame labels* that are unique identifying names of adapters and help localize the fault at hand. An adapter's blame label should uniquely identify the deployment of the adapter's enhanced service. In some settings, this may be as simple as the hostname of the service.

Each adapter has local state that contains sufficient information to determine which messages to intercept and which contracts to enforce on these messages. The local state consists of a mutable *blame registry* and *confirmation mapping*.

- The blame registry tracks contract information about services and requests and who is responsible for this contract information. It maps *service entries* and *request entries* to *blame information*.
  - Service entries track contract information about services: a service entry is a pair of an endpoint and an indexed contract.
  - Request entries track contract information for requests made by and received by the Whip-enhanced service: a request entry is a pair of a unique request identifier and the service entry to which the request was made.
  - Blame information consists of blame labels that identify a set of adapters as the sources of the assumption that the relevant service should satisfy the indexed contract. When a contract violation occurs, blame information is used to generate log messages to help determine why the contract failed. The fine granularity of service entries and request entries allows Whip to precisely track blame information, and produce useful log messages.
- The confirmation mapping tracks whether a service has in fact agreed to implement a contract family k, or if this is merely asserted by a third party. The confirmation mapping maps service entries and request entries to their *confirmation status*. If a service entry or request entry is *confirmed*, then the adapter can correctly assume that the endpoint for the appropriate service has committed to the associated indexed contract.

The local state of each adapter is initialized with information about well-known service endpoints that the Whip-enhanced service might communicate with, what contracts those communications should be held to, and whether the endpoints are confirmed. Any requests to those configured endpoints are intercepted and checked against the specified contract. The local state is updated as the adapter processes messages it intercepts and learns about new services, observes requests, and finds out that services are confirmed. An adapter uses its state to lookup the contract family it believes a host adheres to and to assign blame in the event of a contract failure.

[6] Extraction of identifiers to associate requests and replies is encapsulated in the wire protocol parsing plugins; most of our Whip implementation does not need to worry about it.

Configuring Whip. We assume that all adapters have access to the same fixed Whip contract specification (as described in Section 2). That is, in our current version of Whip, adapters can learn about new services, but not about new contract families. This is not a fundamental restriction.

In addition, adapters are given a configuration file that describes, for each contract family, the low-level wire protocol used and the syntactic representation of messages for this contract. This enables the adapter to parse the raw bytes comprising a message and extract values needed for contract checking and identifying new service entries.

The separation of the syntactic interface of a service and its contract allows Whip to handle multiple wire protocols and RPC frameworks within a single distributed application, and even within a single Whip-enhanced service. This is a necessity due to the loosely-coupled and heterogeneous nature of modern services. Currently, Whip supports SOAP (defined by a WSDL), REST, and Thrift messages. Support for more protocols can be added without modifications to the design of Whip or its contract language.

Although the design and implementation of this configuration and parsing information is one of the significant engineering challenges we encountered—and essential to developing a useful and practical tool—for the rest of the paper we focus primarily on how we track contract and blame information and enforce contracts.

3.2 Enforcing Contracts

In this section we describe how an adapter uses its local state to check the contracts from Section 2.

When a service makes a request, the adapter intercepts the message if the destination endpoint matches a service entry's endpoint in the adapter's blame registry. Alternatively, if the service is making a reply, the adapter intercepts the message if the destination endpoint matches a request entry's endpoint in the adapter's blame registry. If a matching entry is found, then it is checked according to the contract family given in the blame registry for the entry. We discuss the behavior of each kind of contract check the adapter performs and then describe the behavior of the adapter when no matching entry is found.

3.2.1 First-order Operation Contracts. The most basic Whip operation contracts consist of pre- and post-conditions that check first-order properties of application message data.

After the message is parsed according to the configured wire protocol, the pre-condition check is made for request messages and the post-condition for reply messages. In the event of a contract failure for a pre-condition, the sending service is always blamed as it is the initiator of the request.

Post-condition checks involve reply messages. Thus the state of the adapter of the enhanced service that performs a post-condition check contains already a request entry for the request that triggered the reply message. If the check fails, Whip logs a contract error with the blame labels from the blame registry for the request entry. We describe in Section 3.3 which blame labels are used for a request entry in an adapter's blame registry, but intuitively, the blame labels are the sources of the assumption that the endpoint satisfies the associated indexed contract.

3.2.2 Higher-order Operation Contracts. As described in Section 2, higher-order operation contracts contain an **identifies** tag.

For a message that invokes an operation that includes an **identifies** tag in its contract, the message contents may reference (zero or more) endpoints that according to the operation's contract should adhere to indexed contracts. For example, in our Evernote example, the reply message for the listLinkedNotebooks operation references noteStoreURL which according to the contract of listLinkedNotebooks should adhere to the indexed contract NoteStore⟨shareKey⟩.

The Whip adapter updates its blame registry to record the newly identified endpoint (and the indexed contract associated with it) so that future messages to and from the endpoint are intercepted

and appropriate contract checks performed. That is, returning to our Evernote example, the adapter updates its blame registry to include a service entry for the endpoint at noteStoreURL that associates the endpoint with the indexed contract NoteStore⟨shareKey⟩.

3.2.3 Bypass Checks. If no matching service entry or request entry is found in the blame registry for an intercepted message, the adapter *bypasses* contract checks for the message. There are two reasons that there is no matching entry in the blame registry: if no contract information is available, or there is *conflicting* contract information for the endpoint. There may be no contract information in the cases when we are interested in checking contracts for only some of the network communication performed by a service. For example, we may choose to ignore web browser requests by the service. The adapter may also bypass the checks if the local blame registry contains a *conflict* for an endpoint. A conflict occurs when there is no confirmed service entry for the endpoint and multiple service entries that disagree on the contract family for the endpoint. Since we require that an endpoint implements at most one contract family, at least one of the unconfirmed service entries is incorrect (but we do not necessarily know which one).

Note that Whip is carefully designed so that if an adapter's state contains a confirmed service entry, then the corresponding endpoint has definitely agreed to the specified contract family. Thus any unconfirmed service entries that disagree with the confirmed contract entry can be ignored. Put another way, unconfirmed service entries are simply "best effort" approaches to identifying the contracts of the corresponding endpoints; they are merely hearsay as they concern claims made by other adapters about what contract a service adheres to. Once the truth is found through a confirmed service entry, Whip ignores the hearsay of unconfirmed service entries and sticks to the contract the endpoint has agreed to honor.

3.3 Tracking Blame

Central to Whip's contract checking mechanism is the blame information it provides upon a post-condition violation. The key intuitive idea is that in the event of a contract violation by a given endpoint, a Whip adapter should blame the service(s) that (from the adapter's perspective) are responsible for the initial association between the endpoint and the contract.

We describe how Whip adapters update their blame registry to record and track blame information in order to achieve this goal. We first introduce some terminology to help describe different ways an adapter updates its blame registry. Given an adapter A, a service contract c, and a message m, m *identifies a service entry* if checking the relevant part of c against m results in the association of an endpoint with an indexed contract via the **identifies** tag of c. If the identified service entries are not already in the blame registry of A, A extends its blame registry accordingly. Given an adapter A and a message m, m *invokes a service entry* if it is a request message to the endpoint of the service entry, and due to processing m, A checks m against the relevant part of the contract that the service entry associates the endpoint with. Note that the relevant service entry may not be in the blame registry of A before the invocation. In fact the invocation may cause A to add the service entry to its blame registry. For instance, in our Evernote example consider that the Client is enhanced and its adapter's blame registry maps Server2 to the indexed service contract NoteStore<k>. If the Client authenticates to Server2 using k' (i.e., a shareKey other than k) then the invocation results in a new service entry in the blame registry of the Client's adapter that associates Server2 with NoteStore<k'>.

Service entries are created and updated in an adapter's blame registry only when the adapter processes messages that identify or invoke service entries. The adapter aims to use the most precise blame information when creating new entries in its blame registry. However this is not always possible. For example, when an enhanced service adds a service entry due to processing a request from a (seemingly) unenhanced service, the blame label of the unenhanced sender of the message

is unknown. In these situations the special blame label † is used to represent unknown blame information. All other blame labels uniquely identify Whip adapters; blame label † can be thought of as representing all services that an adapter believes are non-Whip-enhanced. Request entries are created when an adapter processes a request message (either as a sender or receiver) and their blame information is determined by the corresponding invoked service entry. Once created, request entries are not modified.[7] We describe further the various ways adapters extend their blame registries with service and request entries focusing on blame information.

- When an enhanced service sends a request message to an endpoint:
  - If the message invokes a service entry, the adapter of the enhanced service may create a new service entry in its blame registry as discussed above. The blame information for such a new service entry is the adapter's own label. Intuitively, this is because the black box of the enhanced service initiated the request, invoking a service entry not previously seen, and so it is solely responsible for the claim that the endpoint should be associated with the corresponding indexed contract.
  - If the message identifies a service entry that is not in the blame registry of the enhanced service's adapter, a new service entry is created. The blame information for such a service is the blame label of the enhanced service itself.
  - If the message identifies a service entry that is in the blame registry of the enhanced service's adapter, the blame information described above is merged (set union) with the existing blame information for the service entry.
  - If the message invokes a service entry, and the service entry is not confirmed, a new request entry is created, and its blame information is the same as the blame information for the invoked service entry. The blame labels for the request entry are used to assign blame in the event of a post-condition contract violation (i.e., the corresponding reply violates its contract). For this reason, the request entry inherits its blame information from the invoked service entry. Put differently, the blame labels are those of the enhanced service(s) that asserted that the endpoint should satisfy the associated indexed contract. In many cases, the enhanced service that made the assertion is in fact the service itself (e.g., through its initial configuration). If the invoked service entry is confirmed then no request entry is created as the adapter does not check a post-condition on the reply message for the request (see Section 3.4, below).
- When an enhanced service receives a request message:
  - If the message invokes a service entry, the adapter of the enhanced service may create a new service entry in its blame registry. If the message originates from another enhanced service's Whip adapter and contains enhanced information (see Section 3.4, below), the blame information for the new service entry is the same as in the sender's blame registry (since the sender's adapter has sent this blame information). Otherwise, since the sender of the message is unknown, the blame information is the special blame label †.
  - If the message identifies a service entry that is not in the blame registry of the enhanced service's adapter, a new service entry is created. If the message originates from another enhanced service's Whip adapter and contains enhanced information, the blame information for the new service entry is the same as in the sender's blame registry (since the sender's adapter has sent this blame information). Otherwise, the blame information is the special blame label †.

[7] In the implementation, request entries exist only for the duration of the request; in the formal model of Section 4, request entries are never removed from the blame registry.

  - If the message identifies a service entry that is in the blame registry of the enhanced service's adapter, the blame information described above is merged (set union) with the existing blame information for the service entry.
  - If the message invokes a service entry, a new request entry is created, and its blame information is the same as the blame information for the invoked service entry.
- When an enhanced service sends or receives a reply message, if the message identifies a new service entry, the service's adapter adds a new service entry to its blame registry whith the same blame information as the blame information of the corresponding request entry.

Back to our example from the beginning of this sub-section, due to Client's request to Server2, Client's adapter adds to its blame registry a service entry whose blame information is the blame label for Client. Since this service entry is unconfirmed, Client's adapter also adds to its blame registry a request entry with the label of Client as its blame information, the same blame information as the newly added service entry. Thus if the Client's adapter discovers that the reply from Server2 violates the corresponding post-condition from the noteStore contract, the adapter logs a contract violation blaming the Client.

Blame for higher-order contracts. Readers may find it surprising that the blame information for an identified service entry introduced by a reply message is the same as the client's blame information for the invoked service entry. However, this is in keeping with the philosophy of higher-order contracts in functional programming languages [Dimoulas et al. 2011; Findler and Felleisen 2002]. This is because a service entry identified in a reply is analogous to a function f returning a closure g; in higher-order contracts for functional programming, the contract to enforce on g is derived from the contract for f, and so the blame labels for g are the same as the blame labels for f.

To make this design decision more concrete, consider a variant of the Evernote example from above. Suppose that Client sends a request to Server1 and from the reply identifies that, according to Server1's contract, Server2 should adhere to the contract NoteStore⟨k⟩. Moreover, assume that Client sends a request to Server2 that invokes the service entry that associates Server2 with NoteStore⟨k⟩, and Client detects that the reply violates the relevant post condition of the contract. Who should be blamed for the violation? Naively we may say Server2. However, from the perspective of Client, Server2 never agreed to adhere to contract NoteStore⟨k⟩. It is Server1 that made an error in asserting that Server2 implements NoteStore⟨k⟩. That is, Server1 did not live up to its contract since listLinkedNotebooks returns a service, Server2, that for whatever reason does not meet NoteStore⟨k⟩. Making sure that Server1 returns a service that meets NoteStore⟨k⟩ is the right fix to the problem at hand. After all, Server2's behavior may be what its other clients expect. The fact that this behavior triggered a contract violation is only because our Client relied on Server1 to live up to its contract. Thus, transitively, Whip blames whoever Client believes is responsible for the assumption that Server1 implements an appropriate contract for listLinkedNotebooks.

3.4 Leveraging Other Adapters

When the destination of a message is known to be Whip-enhanced, an adapter enriches the message with extra information to help the receiver adapter improve the accuracy of its blame information. This *enhanced interaction* differs from *unenhanced interaction* as enhanced interaction includes additional information understandable only by adapters, whereas unenhanced interaction is equivalent to the messages sent by the black boxes.

Enhanced messages are sent only between adapters, as we cannot assume that black boxes understand them. An enhanced message attaches to a black box message. Enhanced messages are translated back to their original black box messages before being passed to a black box.

$$
\begin{array}{lrcl}
\textbf{Processes} & P, Q & ::= & \boxed{a} \mid m \texttt{ to } a \mid P \parallel Q \\
\textbf{Messages} & m & ::= & \texttt{req } \#n \texttt{ from } a : s \\
 & & \mid & \texttt{reply } \#n \texttt{ from } a : s \\
\textbf{Host names} & a, b & \in & \mathcal{A} \\
\textbf{Identifiers} & n & \in & \mathcal{N} \\
\textbf{Bit strings} & s & \in & \{0,1\}^* \\
\textbf{Contexts} & \mathcal{P} & ::= & [\cdot] \mid \mathcal{P} \parallel P
\end{array}
$$

Send-Request

$$\frac{n \text{ fresh} \qquad a \neq b}{\mathcal{P}[\boxed{a}] \longrightarrow \mathcal{P}[\boxed{a} \parallel \texttt{req } \#n \texttt{ from } a : s \texttt{ to } b]}$$

Send-Reply

$$\frac{a \neq b}{\mathcal{P}[\boxed{a}] \longrightarrow \mathcal{P}[\boxed{a} \parallel \texttt{reply } \#n \texttt{ from } a : s \texttt{ to } b]}$$

Receive-Message

$$\frac{}{\mathcal{P}[\boxed{a} \parallel m \texttt{ to } a] \longrightarrow \mathcal{P}[\boxed{a}]}$$

Fig. 6. Core *WM* syntax and reduction rules

Enhanced messages contain information from the sender's blame registry for relevant service entries and request entries. This blame information is used as described in Section 3.3 above. Enhanced messages also contain confirmation status of service entries and request entries, to propagate knowledge of confirmed services (i.e., services known to be Whip-enhanced).

Whip favors enhanced interaction as this helps make blame information more precise. However, enhanced interaction can occur only when the sender knows (based on confirmation status) that the receiver is Whip-enhanced. Enhanced interaction may not be possible due to partial deployment (i.e., one of the black boxes does not have a Whip adapter) or because the two communicating Whip-enhanced services are not aware that the other is also Whip-enhanced (which may occur if the initial confirmation mapping in the adapter's local state did not confirm the other adapter, and previous enhanced messages in the system have not propagated confirmation of the other adapter). Figure 5 depicts enhanced interaction between two adapters (with blue arrows) together with unenhanced interaction with service that is not Whip-enhanced (with black arrows).

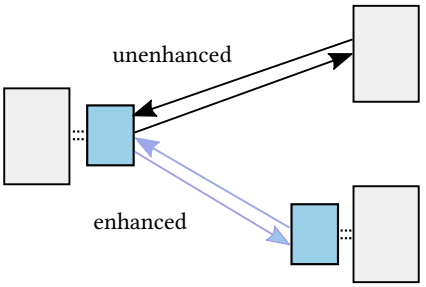

Fig. 5. Whip-enhanced interaction

Additionally, when two Whip-enhanced services use enhanced interaction, they share the burden of contract checks; the sender of the request checks the pre-conditions, and the sender of the reply checks the post-conditions. Conversely, when adapters use unenhanced interaction, each adapter performs both pre- and post-condition checks, in case the other side is not enhanced.

4 WHIP FORMALLY

In this section, we gradually introduce *WM*. We first describe Core *WM*, a model of how modern services interact (Section 4.1). We extend Core *WM* to *WM*, a model for Whip (Section 4.2). We precisely describe the state of the adapter (Section 4.3), how it uses its state to intercept and check messages for contract violations (Section 4.4), and how it updates its state (Section 4.5). We formally show Whip assigns blame correctly in Section 5. *WM* demonstrates formally how Whip meets the requirements laid out in Section 1.

4.1 Core *WM*: Distributed Black Boxes

Figure 6 shows the syntax and reduction rules of Core *WM*, which captures communication between black-box services. Processes P and Q represent black-box services and the messages they exchange. There are two kinds of basic processes: black boxes $\boxed{a}$ and messages m. A compound process

$P \parallel Q$ represents the parallel composition of processes P and Q. We assume standard structural equivalence.

A black box $\boxed{a}$ represents a service with host name a. To capture the lack of access to a service's source code, black boxes are opaque and we can observe only messages they send and receive. For example, the client and two note stores from Section 2 are modeled as black boxes, which we refer to as $\boxed{\texttt{client}}$, $\boxed{\texttt{server1}}$, and $\boxed{\texttt{server2}}$.

Messages are of the form m `to` a where a denotes the name of the recipient of the message. In Core *WM* there are two types of messages: *requests* and *replies*. Request messages are of the form `req` $\#n$ `from` $a : s$ and reply messages are of the form `reply` $\#n$ `from` $a : s$, where a is the originator of the message, s is the payload of the message,[8] and n is the *request identifier*. A request identifier uniquely matches a request with its reply message.

In Core *WM*, processes evolve when black boxes consume and spawn messages. The reduction rules are of the form $P \longrightarrow P'$. Rule Send-Request in Figure 6 shows that $\boxed{a}$ can produce a request message `req` $\#n$ `from` $a : s$ using a *fresh*[9] request identifier n. Rule Send-Reply shows that $\boxed{a}$ can spawn a reply message for any request identifier n. In practice, though, a meaningful request identifier in a reply message would come from a previous request it received. Finally, rule Receive-Message shows how a black box with host name a can consume request and reply messages.

Returning to the example from Section 2, we show the trace of the first request for listLinkedNotebooks between client and server1 (1), the client making the request (2), server1 consuming the request (3), server1 responding to the request (4), and client consuming the request (5):

$$\boxed{\texttt{client}} \parallel \boxed{\texttt{server1}} \longrightarrow \tag{1}$$

$$\boxed{\texttt{client}} \parallel \texttt{req } \#n_1 \texttt{ from client : 'list...' to server1} \parallel \boxed{\texttt{server1}} \longrightarrow \tag{2}$$

$$\boxed{\texttt{client}} \parallel \boxed{\texttt{server1}} \longrightarrow \tag{3}$$

$$\boxed{\texttt{client}} \parallel \boxed{\texttt{server1}} \parallel \texttt{reply } \#n_1 \texttt{ from server1 : '[share...' to client} \longrightarrow \tag{4}$$

$$\boxed{\texttt{client}} \parallel \boxed{\texttt{server1}} \tag{5}$$

4.2 Adding Whip Adapters

Equipped with a model of modern service interaction, we extend *WM* with Whip adapters. Conceptually, an adapter wraps around a black-box service. In *WM* (and Whip), adapters are mutually-trusted. To reflect the independent deployability of modern services, we do not model adapters with access to a shared state. Adapters piggyback on service messages to update each other's local state. Though it adds complexity, we formally model this explicit state transfer as it is necessary to efficiently and precisely blame contract violators.

[8] We included the bit strings in the formalism to capture the opacity of message payloads: interpreting message payloads is protocol- and/or application-specific. That is, a message can be properly interpreted only in the context of a particular contract.

[9] To ensure that request identifiers are locally unique, the black box draws in succession fresh identifiers from a local enumerable set. Locally unique request identifiers together with globally unique host names guarantee that each request is globally unique.

Figure 7 presents the syntax of *WM*, extending the syntax of Core *WM*. An adapter process $\texttt{mon}^l(\sigma, P^a)$ wraps a *base process* P^a, which consists of a black box, and a (possibly empty) list of Core *WM* messages that the black box has just sent and have yet to be processed by the adapter, or that the adapter has forwarded to the black box. Each adapter has a unique label l and local state σ that contains information for checking contracts and assigning blame. Back to the example from Section 2, we can model the interaction of a Whip-enhanced client with a Whip server1, and a non Whip-enhanced server2 as: $\texttt{mon}^{l_c}(\sigma_c, \boxed{\texttt{client}}) \parallel \texttt{mon}^{l_s}(\sigma_s, \boxed{\texttt{server1}}) \parallel \boxed{\texttt{server2}}$.

Processes	P	$::= \ldots \mid \texttt{mon}^l(\sigma, P^a) \mid \widehat{m}\ \texttt{to}\ a$
Base Processes	P^a	$::= \boxed{a} \mid P^a \parallel m\ \texttt{to}\ b$
Enhanced Messages	$\widehat{m}$	$::= m\ \texttt{with}\ \{\texttt{se-blame:=}\widetilde{l};\ \texttt{id-blame:=}\widetilde{l};\ \texttt{id-conf:=}c\}$
Log Entries	le	$::= \texttt{Pre}(se, l) \mid \texttt{Post}(se, \widetilde{l})$
Service Entries	se	$::= a\ \texttt{satisfies}\ k\langle \mathsf{v} \rangle$
Request Entries	re	$::= \#n\ \texttt{from}\ a\ \texttt{expects}\ se$
Contract Names	k	$\in \mathcal{K}$
Blame Labels	l	$\in \mathcal{L}$
Contract Indices	v	$\in \mathcal{V}$

Fig. 7. *WM* syntax

4.3 Adapter State

Section 3.1 presents informally the state of Whip adapters, which we map here to their formal counterparts in *WM*. Each adapter maintains local state σ, for which the syntax is given in Figure 8. Please ignore portions of the Figure shaded in gray; they will be discussed in Section 5.

State	σ	$::= (S, B, C, \widetilde{le}, V)$
Specification	S	$: k \mapsto (e, e)$
Predicates	e	$: m \mapsto \text{bool}$
Blame Registry	B	$: (se \mapsto \widetilde{l}) \cup (re \mapsto \widetilde{l})$
Confirmation Map	C	$: (se \mapsto c) \cup (re \mapsto c)$
Confirmation	c	$::= \checkmark \mid \text{✗}$
Provenance Map	V	$::= se \mapsto pe$
Provenance Entry	pe	$::= a\ \textbf{intro} \mid se\ \textbf{intro} \mid \textbf{learned}$

Fig. 8. *WM* store syntax

State σ is a tuple $(S, B, C, \widetilde{le})$, consisting of its specification S, blame registry B, confirmation map C and error log $\widetilde{le}$. For brevity, we write $\texttt{spec}_\sigma$, $\texttt{blame}_\sigma$, $\texttt{conf}_\sigma$, and $\texttt{errors}_\sigma$ to project each element, respectively, of state σ.

The first piece of state is the specification $\texttt{spec}_\sigma = S$ that maps contract names k to pre- and post-conditions for a service's operation. For simplicity, we assume that every service offers one operation. As a result of this simplification, a service contract is made up of exactly one operation contract and so, in this section, we refer to both service contracts and operation contracts as contracts. A specification S maps contract names to contracts. Formally, S maps a contract name k to a pair (e_{pre}, e_{post}). We leave the syntax of predicates e unspecified, but require them to be total (i.e., terminating) single argument boolean functions. Specifications do not change during execution, nor are transferred between adapters. Moreover we assume specifications contain information about *all* contract names used by an adapter.

The second piece of state is the blame registry $\texttt{blame}_\sigma = B$, which maps service entries and request entries to blame information. A service entry $se = a\ \texttt{satisfies}\ k\langle \mathsf{v} \rangle$ indicates that the service at host a should implement indexed contract $k\langle \mathsf{v} \rangle$. As a strict design rule, we require that a host implements at most one contract family. A request entry $re = \#n\ \texttt{from}\ a\ \texttt{expects}\ se$ indicates that host a made a request with request identifier n to a service with service entry se. Entries in the blame registry of a Whip-enhanced service a correspond to assumptions that a has about other services, and that clients of a have about a. Blame information $\widetilde{l}$ is the set of labels of adapters

Enhanced Interaction

ENHANCED-SEND
$$\frac{(k, \checkmark) = \texttt{contract\_for}_\sigma(b, m) \quad \sigma', \widehat{m} = \texttt{lift}_\sigma(k, \checkmark, m, l)}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a \parallel m \ \texttt{to}\ b)] \rightarrow \mathcal{P}[\texttt{mon}^l(\sigma', P^a) \parallel \widehat{m} \ \texttt{to}\ b]}$$

ENHANCED-RECEIVE
$$\frac{(k, \checkmark) = \texttt{contract\_for}_\sigma(a, \widehat{m}) \quad \sigma', m = \texttt{lower}_\sigma(k, \checkmark, \widehat{m})}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a) \parallel \widehat{m} \ \texttt{to}\ a] \rightarrow \mathcal{P}[\texttt{mon}^l(\sigma', P^a \parallel m \ \texttt{to}\ a)]}$$

Unenhanced Interaction

UNENHANCED-SEND
$$\frac{(k, \text{✗}) = \texttt{contract\_for}_\sigma(b, m) \quad \sigma', \widehat{m} = \texttt{lift}_\sigma(k, \text{✗}, m, l) \quad \sigma'', m = \texttt{lower}_{\sigma'}(k, \text{✗}, \widehat{m})}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a \parallel m \ \texttt{to}\ b)] \rightarrow \mathcal{P}[\texttt{mon}^l(\sigma'', P^a) \parallel m \ \texttt{to}\ b]}$$

UNENHANCED-RECEIVE
$$\frac{(k, c) = \texttt{contract\_for}_\sigma(a, m) \quad \sigma', \widehat{m} = \texttt{lift}_\sigma(k, \text{✗}, m, \dagger) \quad \sigma'', m = \texttt{lower}_{\sigma'}(k, \text{✗}, \widehat{m})}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a) \parallel m \ \texttt{to}\ a] \rightarrow \mathcal{P}[\texttt{mon}^l(\sigma'', P^a \parallel m \ \texttt{to}\ a)]}$$

Bypass Adapter

BYPASS-SEND
$$\frac{\texttt{contract\_for}_\sigma(b, m) \text{ undefined}}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a \parallel m \ \texttt{to}\ b)] \rightarrow \mathcal{P}[\texttt{mon}^l(\sigma, P^a) \parallel m \ \texttt{to}\ b]}$$

BYPASS-RECEIVE
$$\frac{\texttt{contract\_for}_\sigma(a, m) \text{ undefined}}{\mathcal{P}[\texttt{mon}^l(\sigma, P^a) \parallel m \ \texttt{to}\ a] \rightarrow \mathcal{P}[\texttt{mon}^l(\sigma, P^a \parallel m \ \texttt{to}\ a)]}$$

Fig. 9. *WM* reduction rules

that introduced the relevant service. In Section 4.5, we describe formally how blame information is propagated to assign blame in the event of contract violations.

An adapter records the confirmation status of each service entry and request entry in its confirmation mapping $\texttt{conf}_\sigma = C$. If a service entry, a `satisfies` $k\langle \mathsf{v} \rangle$, or a request entry, `#`n `from` b `expects` a `satisfies` $k\langle \mathsf{v} \rangle$, is confirmed (i.e., confirmation status is ✓), then a is enhanced (i.e., is wrapped by an adapter), and the adapter state of a also associates the service it offers with the contract name k.

Finally, an adapter's local state contains a set of log entries $\texttt{errors}_\sigma = \widetilde{le}$. Each log entry records the failure of a contract. A pre-condition log entry $\texttt{Pre}(a \ \texttt{satisfies}\ k\langle \mathsf{v} \rangle, l)$ indicates that the black box wrapped with the adapter with label l violated the pre-condition for indexed contract $k\langle \mathsf{v} \rangle$ while making a request to a. A post-condition log entry $\texttt{Post}(a \ \texttt{satisfies}\ k\langle \mathsf{v} \rangle, \widetilde{l})$ indicates that black box a sent a reply for a request it received but the reply failed to meet the post-condition of indexed contract $k\langle \mathsf{v} \rangle$, and that the adapters with a label $l \in \widetilde{l}$ are to blame, i.e., they are responsible for the assumption that a would satisfy the indexed contract.

4.4 Adapter Message Interception and Contract Checking

Before delving into the semantics of *WM*, we first introduce formally *enhanced messages*, the messages adapters exchange in enhanced interaction (see Section 3.4). An enhanced message m `with {` `se-blame:=`$\widetilde{l}$`; id-conf:=`c`; id-blame:=`$\widetilde{l_{id}}$ `}` attaches to a black box message m the additional information that the receiver should hold $\widetilde{l}$ responsible if m does not live up to its contract. The enhanced message also contains blame information $\widetilde{l_{id}}$ and confirmation status c for the identified service in the message. For example, `server1`'s reply to the client, `reply #`n `from server1 : '[share...' to client`, would identify `server2 satisfies NoteStore⟨shareKey⟩` and the confirmation status c in the enhanced message would indicate whether `server2` is known to satisfy contract `NoteStore⟨shareKey⟩` and further that `server2` is Whip-enhanced. To simplify the model, each message's payload identifies a single service.

The reduction semantics of *WM* include the reductions of Core *WM* (which allow black boxes to consume and spawn messages). All messages sent to or from a wrapped black box are intercepted by the adapter, which processes the message (possibly performing contract checks and/or updating the adapter's internal state) and then forwards the message onward (possibly transforming an unenhanced message to an enhanced message, or vice-versa).

There are six rules shown in Figure 9, in three groups: (1) Enhanced Interaction (cf. Section 3.4); (2) Unenhanced Interaction (cf. Section 3.4); and (3) Bypass Adapter (cf. Section 3.2.3). Within each group, there is one rule for when the wrapped black box is sending a message, and one rule for

```
1:  function lift_σ(k, c, m, l)
2:    (se, re, se_id) = entries(m, k)
3:    update-lift_σ(m, c, se, re, se_id, l) ▷ State update
4:    (pre, post) = spec_σ(k) ▷ Fetch contract predicates
5:    if type(m) == req then ▷ Request sent
6:      if ¬pre(m) then errors_σ += Pre(se, l)
7:    else ▷ Reply sent
8:      l̄ = blame_σ(re) ▷ Fetch recorded blame for re
9:      if ¬post(m) then errors_σ += Post(se, l̄)
10:   return m with {se-blame = blame_σ(se);
                     id-blame = blame_σ(se_id);
                     id-conf = conf_σ(se_id)}

11: function lower_σ(k, c, m̂)
12:   (se, re, se_id) = entries(raw(m̂), k)
13:   update-lower_σ(m̂, c, se, re, se_id) ▷ State update
14:   return raw(m̂) ▷ Return unenhanced part

15: function entries(m, k)
16:   if type(m) == req then ▷ Is request message
17:     (a, b) = (to(m), from(m)) ▷ To a, from client b
18:   else ▷ Is reply message
19:     (a, b) = (from(m), to(m)) ▷ From a, to client b
20:   return (a satisfies k⟨index(m)⟩,
            reqid(m) from b expects (a satisfies
              k⟨index(m)⟩),
            identified(m))
```

Fig. 10. `lift` and `lower` metafunctions.

when the wrapped black box is receiving a message. The decision for which of the six rules to use is based on the internal state of the adapter, the destination of the message, and whether the message is enhanced or not. We discuss each of the three groups in turn.

Metafunction `contract_for`[10] provides convenient access to confirmation status, and is how the adapter chooses to use one of these three groups. Conceptually, `contract_for` searches the state for a service entry or request entry whose host matches the destination of the message and returns the contract family and confirmation status of that entry. It is possible that `contract_for` is undefined; we return to this last case in Section 4.4.3.

4.4.1 Enhanced Interaction. The rules in this group apply when the recipient of the message is known to be Whip-enhanced, i.e., $\texttt{conf}_\sigma$ contains a mapping for a service entry for the message recipient that maps to a confirmed ✓ status. This is captured by metafunction `contract_for` returning $(k, ✓)$ where k is the contract name for the confirmed service. Returning to the example of Section 2, if `client` sent a message to `server1`, i.e., $\texttt{mon}^{l_c}(\sigma_c, \texttt{client} \parallel m \texttt{ to server1})$, the enhanced send rule would be used if server1 was known by client's adapter to be Whip-enhanced: $\texttt{contract\_for}_{\sigma_c}(\texttt{server1}, m) = (\texttt{NoteStore}, ✓)$.

When the recipient of the message is known to be Whip-enhanced (i.e., confirmed), the adapter transforms the message by "lifting" it to an enhanced message via the `lift` function. On the receiving end, the recipient adapter will "lower" the enhanced message it received back to an unenhanced message via the `lower` metafunction. Figure 10 presents the definitions of `lift` and `lower` as pseudocode. Both metafunctions imperatively update the adapter's state σ as a result of contract checking, confirmation propagation, and blame propagation. (We defer explaining the state update functions until Section 4.5.) We first describe `lift` and then `lower`.

Metafunction $\texttt{lift}_\sigma$ takes as arguments: the name of the contract to check the message against (k); confirmation status of the other communication party (c); the intercepted message to "lift" (m); and the label of the adapter of the sender of the message (l).[11] It returns a tuple $(\sigma', \widehat{m})$ where σ' is the implicitly returned final updated state of the adapter, and $\widehat{m}$ is the transformed enhanced message. We now describe each sub-task of `lift`.

Extract contract information. `lift` extracts the contract information from the message it is processing using the `entries` metafunction to produce the service entry for the message, the request entry, and service entry of the identified service in the message. Internally it uses helper

[10] Defined formally in Appendix A.1.

[11] For Enhanced-Send, this is the adapter's own label.

metafunctions to parse the message: $\texttt{from}(m)$ extracts the origin of the message m and $\texttt{to}(m)$ extracts the destination of the message m. For simplicity we assume two *message parsing* metafunctions that can parse Whip-specific information from a message payload: $\texttt{index}(m)$ is the index for the contract family and corresponds to the tag **where index is** in the Whip IDL; `identified` corresponds to the result of the **identifies** tag, and indicates that the payload *identifies* a service that should implement a certain indexed contract. Returning to the running example of Section 2, server1's reply message to client's request for listlinkedNotebooks is parsed as:

```
entries_σs(reply #n from server1 : '[(share...', NoteStore) = (
        server1 satisfies NoteStore⟨v⟩,
        #n from client expects server1 satisfies NoteStore⟨v⟩,
        server2 satisfies NoteStore⟨shareKey⟩)
```

In this example v is the index used when the client called listLinkedNotebooks on server1.

Pre-condition checks. Upon a contract violation, the `lift` metafunction constructs a log entry deriving blame from its arguments. For rule Enhanced-Send (Figure 9), these arguments come from the label of the adapter. That is, for failure of a pre-condition, the service sending the request (hereafter the client) is always blamed.

Post-condition checks. Post-condition violations occur for the service sending the reply (hereafter the server). to the client. Whip logs a contract error with the blame labels $\widetilde{l}$ from the server's blame registry for the request entry re. We discuss in more detail formally how blame information is transferred in Section 4.5.

Message transformation. The final step of `lift` is to construct a new enhanced message (line 10) to transfer information from the local adapter's state to the receiving adapter. The enhanced message includes the client's blame information for the service entry, blame labels for the service entry that was identified in m, and the client's confirmation status recorded for the identified service. This enhanced message is then sent to the receiving adapter using the Enhanced-Send rule. The enhanced message is processed by the Enhanced-Receive rule and transformed back to an unenhanced message via the `lower` function, which we now describe.

Metafunction $\texttt{lower}_\sigma$ takes the following arguments: the name of the contract to check the message against (k); whether the sender of the message is confirmed (c); and the intercepted enhanced message to "lower" ($\widehat{m}$). It returns a tuple (σ', m) where σ' is the implicitly returned final updated state of the adapter, and m is the transformed unenhanced message.

Whereas `lift` performed contract checks and introduced new blame for service entries, `lower` only updates its internal state based on the state transferred by the sending adapter. After the adapter state is updated, the enhanced message is transformed to its unenhanced counterpart via the `raw` function which simply discards the enhanced message metadata, i.e., if $(\sigma', \widehat{m}) = \texttt{lift}_\sigma(k, c, m, l)$ then $\texttt{raw}(\widehat{m}) = m$.

4.4.2 Unenhanced Interaction. Rule Unenhanced-Send applies when an adapter intercepts a message sent by the black box it wraps to host b where $\texttt{contract\_for}_\sigma(b, m) = (k, \not\checkmark)$, meaning the destination of the message is not known to be enhanced.[12] Rule Unenhanced-Receive fires when the adapter intercepts an unenhanced message intended for the the black box it wraps, and the black box's host b should implement contract family k.

[12]Note that it may be the case that the other host is Whip-enhanced, but this fact is not locally known.

```
1:  function update-lift_σ(m, c, se, re, se_id, l)
2:    if type(m) == req then                          ▷ Request sent
3:      blame_σ[se] ←? {l}                            ▷ Initialize blame for se
4:      blame_σ[se_id] ←? {l}                         ▷ Initialize blame for invoked
5:      prov_σ[se] ←? from(m) intro
6:      prov_σ[se_id] ←? from(m) intro
7:      conf_σ[re] ← c                                ▷ Record confirmation for request
8:    else                                            ▷ Reply sent
9:      blame_σ[se_id] ←? blame_σ(re)                 ▷ Init blame for id
10:     prov_σ[se_id] ←? se intro
11:   conf_σ[se_id] ←? ✗                              ▷ Initialize confirmation for id

12: function update-lower_σ(m̂, c, se, re, se_id)
13:   blame_σ[se_id] ←? ∅                            ▷ Initialize blame for id
14:   blame_σ[se_id] ← blame_σ(se_id) ∪ m̂.id-blame
15:   prov_σ[se_id] ←? learned
16:   if m̂.id-conf == ✓ then                         ▷ Id is confirmed
17:     conf_σ[se_id] ← ✓                           ▷ Promote to confirmed
18:   else                                          ▷ Identified not known to be confirmed
19:     conf_σ[se_id] ←? ✗                          ▷ Initialize unconfirmed
20:   if type(m̂) == req then                        ▷ Receive request
21:     blame_σ[re] ← m̂.se-blame                    ▷ Blame for request
22:     conf_σ[re] ← c                              ▷ Confirmation for request
```

Fig. 11. State Update functions

In either case, the adapter takes a best-effort approach to perform contract checking and provide as precise as possible blame information. Specifically, the adapter performs the contract checking and blame propagation that the adapters of the source and the destination of a message *would have* performed if they had opted for enhanced interaction. Thus, in both rules, we see that the adapter uses both metafunctions `lift` and `lower`, to emulate enhanced interaction. The confirmation status argument is ✗ so that the adapter will send an unenhanced reply for an unenhanced request, as the receiver may not be able to interpret an enhanced message.

One key difference between rule Unenhanced-Receive and the two enhanced rules, is that we use the unknown label † as the label for the sender when calling the two metafunctions, instead of the adapter's own label. This is because the adapter, as the recipient of the message, is *not* responsible for its contents. As mentioned in Section 3.3, the base label † is thus used as the label for all non-Whip-enhanced services, and when the label of the sender's adapter is not known.

4.4.3 Bypass Adapters. These rules apply in cases where the adapter chooses not to intercept messages, and so the messages bypass the adapter. This occurs when a Whip-enhanced service communicates with a host b for which there is either no contract information available or conflicting contract information, i.e., $\texttt{contract\_for}_\sigma(b, m)$ is undefined. The messages are not intercepted by the adapter. Messages bypass the adapter rather than getting stuck so that the presence of the adapter does not disturb traffic to and from the black box.

4.5 Updating State and Assigning Blame

In this section we make formal the discussion from Section 3.3 about how adapters update their state when they receive messages and how they keep track of blame information.

All parts of the state (except the immutable specification) are updated as the adapter intercepts and receives messages. State is updated via the `update-lift` and `update-lower` metafunctions, called from the `lift` and `lower` metafunctions, respectively. The update functions take the following inputs as arguments: the message intercepted m or $\widehat{m}$, the confirmation status c of the other communication party, the service entry of the request se, the request entry re, and the service identified in the message se_{id}. In the case of `update-lift`, it also contains the blame label of the sender of the message.

The update functions add new mappings using notation $f_\sigma[k] \leftarrow v$ where f is one of the store projection functions. For example, the result of $\texttt{blame}_\sigma[se] \leftarrow v$ is a modified state σ' where the blame registry contains a mapping $se \mapsto v$. The final updated state σ' is implicitly returned as the first part of the result. We also use an optional assignment syntax $f_\sigma[k] \overset{?}{\leftarrow} v$ that adds mapping

$f_\sigma[k \mapsto v]$ only if $k \notin f_\sigma$. Much of the subtlety of the `lift` and `lower` metafunctions is due to the propagation of blame information which is necessary for correct and precise blame assignment. We first describe `update-lift` and then `update-lower`.

Figure 11 describes the behaviors of the `update-lift` and `update-lower` metafunctions, given as pseudocode. Conceptually, `update-lift` introduces blame information. In essence, it implements the informal description of tracking blame information in Section 3.3. There are two ways a service entry can be introduced into an adapter's blame registry. First, if the black box is sending a request, then both the invoked service entry and the identified service entry may be new to the adapter. That is, the adapter has not previously associated these services with indexed contracts. If so, the adapter uses its own label as the blame label for the new service entries. This can be seen in lines 3–4.

Second, if the black box is sending a response, then the identified service entry may be new to the adapter (lines 8–9). As discussed in Section 3.3, the blame labels for the identified service entry are the blame labels for the corresponding request entry.

Additionally, `update-lift` records the confirmation status for the request entry (line 7). That is, the adapter will send an enhanced reply if and only if it received an enhanced request.

Whereas `update-lift` introduces new blame for service entries, `update-lower` only records and merges the blame and confirmation information from the enhanced message it received. In particular, `update-lower` performs the following sub-tasks:

- The blame labels for the identified service entry from the sender's enhanced message are merged with any existing blame information the receiver had for the identified service entry (lines 13–14).
- Confirmation status for the identified service is merged. If the sender knows the identified service is confirmed then the message will contain a confirmation status of ✓ and the receiver will update its state to record that confirmation (lines 16–19).
- When processing a request, the server creates a request entry for the client request, recording the client's service entry blame (which is equal to the client's request entry blame). Confirmation information for the request entry is also recorded so that the `contract_for` metafunction will be defined when the server sends a reply message (lines 20–22), and ensuring that the reply message will be enhanced if and only if the request message was enhanced.
- The enhanced message is transformed to its unenhanced counterpart via the `raw` function which simply discards the enhanced message metadata (line 14).

5 CORRECT BLAME

In this section, we establish the key metatheoretic result of *WM*: *correct blame assignment*.[13] The blame assignment provided by Whip is a very important part of its usefulness. To show that Whip blames appropriately, we formalized its semantics to show correct blame assignment. Informally, Dimoulas et al. [2011] define that a contract system assigns blame correctly if, given a value that violates a contract, it blames the component that vouched that the value meets that contract. They extend their contract calculus with provenance information and prove that the blame label reported upon a contract violation matches the provenance of the value that triggered the violation. The provenance information is not used in contract enforcement, but provides an intuitive formal basis for specifying blame correctness.

We use the same approach and extend the semantics of *WM* to track provenance. This extension is straightforward, and the tracking of provenance is meant to be obviously correct. The tracking of provenance is independent of the creation and propagation of blame information, and provides a

[13] Complete formalisms and proofs are in Appendix A.

sound basis to specify correct blame. Due to the black-box nature of services in *WM*, we cannot use the provenance tracking mechanism of Dimoulas et al. Instead, each adapter records in its local state provenance information about service entries that reflects how the adapter's registry was updated. Provenance information allows us to easily detect which adapters are responsible for introducing which service entries, and, transitively, for service entries introduced due to the reply from a service with service entry se, which adapters are responsible for introducing se.

The portions of Figures 8 and 11 shaded in gray extend *WM* to track provenance. The local state of an adapter is extended to include *provenance map* V, which maps service entries to *provenance entries*. Intuitively, an adapter's provenance map records for each service entry in the registry how the service entry was added to the registry. If the adapter learned about the service entry se from another service, then $V(se) = \textbf{learned}$. If information about se was introduced because host a sent a request, then $V(se) = a\ \textbf{intro}$. (Host a is typically the host wrapped by the adapter, but due to interaction with non-Whip-enhanced services, it may differ; see rule Unenhanced-Receive.) If service entry se_{id} was introduced due to service entry se identifying se_{id} in its reply then $V(se_{id}) = se\ \textbf{intro}$. The three provenance entries mirror the three ways that service entries can enter an adapter's blame registry, described in Section 4.4.1. (i.e., learned, introduced by a request, or introduced by a reply).

We designed *WM* so that local blame information is, in essence, a summary of provenance information. We express this via a *blame consistent with provenance* judgment $P \Vdash \textbf{blame}\ l\ \textbf{for}\ se$. Intuitively, if there is a post-condition violation that involves a service entry se, and $P \Vdash \textbf{blame}\ l\ \textbf{for}\ se$ holds, then the provenance of se ultimately goes back to l, and so blaming l is consistent with the provenance information.

More generally, if $P \Vdash \textbf{blame}\ l\ \textbf{for}\ se$ holds, then either l is responsible for introducing se, or se was introduced by a response from a service with service entry se' and $P \Vdash \textbf{blame}\ l\ \textbf{for}\ se'$. The following rules for the judgment are the base cases for, respectively enhanced interaction and unenhanced interaction, corresponding to host a introducing service entry se due to sending a request:

$$\frac{\mathsf{prov}_\sigma(se) = a\ \textbf{intro}}{\mathcal{P}[\mathsf{mon}^l(\sigma, P^a)] \Vdash \textbf{blame}\ l\ \textbf{for}\ se} \qquad \frac{\mathsf{prov}_\sigma(se) = b\ \textbf{intro} \quad b \neq a}{\mathcal{P}[\mathsf{mon}^l(\sigma, P^a)] \Vdash \textbf{blame}\ \dagger\ \textbf{for}\ se}$$

The second base case corresponds to when unenhanced interaction means that an adapter can not know the precise provenance (or blame) for service entry se. This is the only case where Whip introduces non-precise blame. In any other case blame information pinpoints accurately a set of black boxes that if a programmer inspects, she will detect the source of the bug.

The inductive case involves blame assigned to a service entry se_{id} that was introduced by the reply from a service entry se. Intuitively, we should blame whoever introduced se, as se_{id} is part of the higher-order result of se. This intuition is captured by the following rule:

$$\frac{\mathsf{prov}_\sigma(se_{id}) = se\ \textbf{intro} \qquad \mathcal{P}[\mathsf{mon}^l(\sigma, P^a)] \Vdash \textbf{blame}\ l'\ \textbf{for}\ se}{\mathcal{P}[\mathsf{mon}^l(\sigma, P^a)] \Vdash \textbf{blame}\ l'\ \textbf{for}\ se_{id}}$$

Blame consistency with provenance, together with certain reasonable assumptions on the initial state of adapters, forms a well-formedness predicate which we show is *preserved* as the process evolves. Well-formedness of *WM* requires that all blame information in an adapter's registry is consistent with provenance, which is sufficient to define *correct blame*. Intuitively, if a well-formed process P takes a step and this step results in an adapter in P detecting a contract violation, then (1) for a violation of a pre-condition, *WM* blames the sender of the request message that caused the violation since they previously "agreed" to the contract by using it; (2) for a violation of a post-condition, *WM* blames consistently with provenance. Formally, our theorem is:

Correct Blame. If well-formed $P_1 = \mathcal{P}_1[\texttt{mon}^l(\sigma_1, P_1^a)]$ and $P_1 \rightarrow P_2$ and $P_2 = \mathcal{P}_2[\texttt{mon}^l(\sigma_2, P_2^a)]$ and $\texttt{errors}_{\sigma_2} = \{le\} \cup \texttt{errors}_{\sigma_1}$ then

(1) if $le = \texttt{Pre}(se, l_e)$, then
  (a) if $P_1 = \mathcal{P}[\texttt{mon}^l(\sigma_1, P^a \parallel m \ \texttt{to} \ b)]$ and
    $P_2 = \mathcal{P}[\texttt{mon}^l(\sigma_2, P^a) \parallel m' \ \texttt{to} \ b]$ then $l_e = l$
  (b) if $P_1 = \mathcal{P}[\texttt{mon}^l(\sigma_1, P^a) \parallel m \ \texttt{to} \ a]$ and
    $P_2 = \mathcal{P}[\texttt{mon}^l(\sigma_2, P^a \parallel m \ \texttt{to} \ a)]$ then $l_e = \dagger$
(2) if $le = \texttt{Post}(se, \widetilde{l})$, then $\forall l \in \widetilde{l}.\ P_2 \Vdash$ **blame** l **for** se.

6 WHIP IN PRACTICE

We have developed a prototype implementation of Whip. It consists of the adapter described in Section 3 (and formalized in Section 4), and an interposition library for redirecting TCP connections through the adapter. The adapter is about 3,800 non-empty lines of Python and the interposition library is about 250 non-empty lines of C.

As described in Section 3, before deployment, users configure Whip adapters with information that describes: (i) what is the contract of the Whip-enhanced service the adapter enhances; and (ii) what are the contracts for other well-known services whose interaction with the Whip-enhanced service the adapter should monitor. Upon deployment, a Whip-enhanced service is linked with the interposition library. At run time, the library intercepts `connect` system calls from the service and contacts the adapter to check whether a new connection should bypass the adapter or not (based on the adapter's blame registry). The adapter's local state (see Section 3) is stored in a disk-backed permanent store, with an in-memory cache for performance. When a cache miss occurs, the requested data is fetched to memory if found. In full generality, garbage collection of adapter state is as hard as determining distributed object lifetimes. In the case studies, however, the lifetimes of the pieces of adapter state are very clearly scoped to an individual user session (defined precisely and differently by each application). Since sessions have precise and finite lifetimes, we feel it would be easy to capture in a configuration directive. We leave garbage collection of on-disk adapter state as future work.

Whip supports any message format given an appropriate message format plugin. We have implemented plugins for Thrift (in 150 lines of code), REST (100 lines) and SOAP (400 lines). To check contracts on encrypted communications (i.e., a service using TLS), the adapter and the service it enhances can share certificates or use a mutually trusted certificate authority to allow the adapter to decrypt messages for the black box. In our prototype implementation Whip acts as a trusted certificate authority. When the black box attempts to access a secure endpoint, Whip uses a certificate for that endpoint signed by the trusted Whip certificate authority. This requires the system administrator to install the Whip certificate as a root of trust in the system.

We have used Whip to harden the interfaces of three real-world off-the-shelf services: Evernote (from Section 2), the Twitter API, and an online correspondence chess service. The complete Whip contracts for the three case studies (and one more) are in Appendices B and C. We discuss the most interesting aspects of the case studies in the remainder of this section, and discuss performance in Section 7.

6.1 Evernote

The Evernote case study showcases four of the aspects of Whip's runtime we discuss in context in Section 1: (i) Whip treats the Evernote server and its clients as black-boxes; (ii) Whip is partially deployed as we cannot enhance the Evernote servers; (iii) Whip does not change communication patterns between the Evernote server and its clients so as not to disrupt their operation; and (iv)

Whip operates both on top of Evernote's Thrift-based API and its simpler HTTPS protocol for OAuth authentication requests.

As in Section 2, we designed Whip contracts for Evernote's API based on its informal documentation. We use *first-order Whip contracts* to express a variety of first-order properties similar to the two first-order properties from Section 2: non-empty strings, bounds checks on integers, malformed GUIDs, strings that are too long, missing parameters that could not be marked as required due to Thrift limitations, and strings not matching certain patterns (e.g., valid MIME type). We use *indexed higher-order Whip contracts* to express properties about the correct use of a multitude of tokens (similar to the second higher-order property in Section 2) despite some of these tokens originating from OAuth rather than Thrift services.

6.2 Twitter

Twitter's REST API[14] allows access to a user's tweets and followers, and is representative of many REST APIs. Its documentation has a series of examples that highlight key properties of the API. We use Whip contracts to turn these examples into a precise and executable specification. Beyond the Evernote case study, the Twitter case study showcases that (i) Whip is compatible with the most popular message format for microservices, REST; and (ii) the Whip contract language allows programmer to write precise contracts with minimum effort reusing code from Python libraries.

First-order Contracts for Well-formed Data. We employed first-order Whip contracts to express a variety of properties of arguments and results of operations of the Twitter API. For example, the operation to fetch tweets must consume either a user ID or a screen name. We encode this disjunctive requirement with a pre-condition. Few API libraries actually defensively check this requirement but instead rely on the server to report back an error message. In addition, we used a post-condition to capture that the result of the operation should be a list of length equal to one of the arguments of the operation (or at most 200 elements).

Some of the properties required careful syntactic checks. Instead of performing these checks ourselves, we leveraged third-party Python libraries to perform the data validation. The Whip contract language allows importing packages via a familiar Python syntax **from** X **import** Y where X is the package name and Y is the name to import. In one case, dates needed to conform to the RFC 822 standard, so the contract imports the parsedate_tz function from the rfc822 Python package.

The following Whip contract language snippet exemplifies how we expressed these properties[15]:

```
from rfc822 import parsedate_tz
service Twitter {
  /1.1/statuses/user_timeline(req)
  @requires « 'user_id' in req.args or 'screen_name' in req.args »
  @ensures «
    assert type(result) == list
    assert ('count' not in req.args or length(result) <= max(200, req.args.count))
    for tweet in result: assert parsedate_tz(tweet.created_at) != None
  »
  ...
}
```

A Higher-order Contract for Valid Tweet IDs. Outside the correct use of OAuth tokens, the correct behavior of Twitter operations depends on the correct use of unique tokens that denote other

[14] https://dev.twitter.com

[15] Assertion failures will result in a contract failure.

types of data, such as tweets. We discuss here an example of such a requirement; retweets should involve tokens that correspond to actual tweets. That is for a request /1.1/statuses/retweet/<id>.json(req), id should be the unique token of an actual tweet. Consequently, a retweet request should use only an id retrieved from a request /1.1/statuses/user_timeline(req) or similar whose reply contains a list of tokens for actual tweets. The following Whip contract expresses this requirement:

```
service Twitter {
    /1.1/statuses/user_timeline(req)
    @foreach tweet in « result» identifies Twitter at receiver with index « 'tweet:' + tweet.id »
    ...

    /1.1/statuses/retweet/<id>.json(req)
    @where index is « 'tweet:' + id »
    @ensures « 'does not exist' not in result.get('errors') »
    ...
}
```

The result of the user_timeline operation identifies that the **receiver** service, i.e., the service that receives a request for this operation, implements contract Twitter⟨'tweet:' + t.id⟩, where t ranges over the tokens in the result of the operation. For a retweet operation, the receiver service must implement contract Twitter⟨'tweet:' + id⟩, where id is part of the request URL. Otherwise, if the post-condition of the operation fails, Whip blames the client for incorrectly claiming that the retweet involved an actual tweet.

A Higher-order Contract for Valid OAuth Tokens. Twitter, like Evernote, uses the OAuth protocol for authentication. The API describes that OAuth tokens passed as arguments should originate from an appropriate OAuth service request. We express the validity of OAuth tokens in a similar manner to the two higher-order properties of Evernote's API from Section 2:

```
from urlparse import parse_qs
service TwitterOAuth {
    /oauth/access_token(req)
    @identifies Twitter at receiver with index « 'oauth:' + parse_qs(result.content).get('oauth_token') »
}
service Twitter {
    /1.1/status/user_timeline.json(request)
    @where index is « 'oauth:' + request['headers'].get('Authorization') »
}
```

We use the parse_qs function from the urlparse package to parse the querystring of the resulting OAuth access token request in order to retrieve the OAuth access token t. The access token is used to identify an indexed contract Twitter⟨'oauth:' + t⟩, which is later used in a subsequent request for the user_timeline operation.

6.3 Xfcc Correspondence Chess

Xfcc is a popular web service (WSDL) specification for correspondence chess.[16] The specification offers a standard for server implementations that manage chess games recognized by the World Chess Federation (FIDE). The specification describes two operations: GetMyGames returns the status of all games the user is playing in, and MakeAMove performs a game action (e.g., move a

[16] http://xfcc.org/

piece, offer a draw). Beyond the Evernote and Twitter case studies, the Xfcc case study showcases that (i) Whip is compatible with the standard message format for traditional web services, SOAP (WSDL); and (ii) Whip is compatible with a diversity of service implementations; (iii) Whip can detect specification violations in both servers and clients; and (iv) indexed contracts can encode complex conditions for the successful call of a service operation.

A First-order Contract for Valid PGN Moves. The GetMyGames operation of Xfcc returns a data structure that represents the status of a game. This data structure includes a moves field that specifies the history of the moves of the game in Portable Game Notation (PGN) format. Similar to the validity of dates in the Twitter case study, we used a third-party library to check the validity of the moves field. The read_game function from the chess.pgn package parses a string containing the list of moves in PGN format and returns a Python structure representing the game. When the parsing fails it throws an exception. With this function in hand, we wrote a contract that ensures that all games are in valid PGN format. The contract succeeds if the read_games function terminates without throwing an exception:

```
from chess.pgn import read_game
service Chess {
    GetMyGames(username, password)
    @ensures «
        for game in result:
            try: read_game(game['moves'])
            except: return False »
    ...
}
```

A Higher-order Contract for Valid Game IDs. The documentation of Xfcc states that when a client provides an invalid game ID to MakeAMove the server should return error code InvalidGameID. Whip can express this property with a contract analogous to the contract for valid tweet IDs in the Twitter case study. We discovered that two popular Xfcc servers return an database error page rather than the documented correct error code. We also found that a popular client was unable to interpret the return code, making an invalid move look successful to its user.

A Higher-order Contract for Accepting a Draw Only When Allowed. The documentation states that draw offers are active only for one move and a player can accept a draw only for a game with an active draw offer. To make a draw offer to an opponent, a player passes **True** as the offerDraw argument of the MakeAMove operation of Xfcc. To accept the draw, the opponent passes **True** as the acceptDraw argument of their immediate next MakeAMove invocation. If a player does not follow the protocol for accepting a draw, the service should return the NoDrawWasOffered error code. Whip can express the compliance of players with the draw protocol with a higher-order indexed contract. This use of indexed contracts differs from those we have seen so far. While in the Evernote and Twitter case studies, we used indices to pair the code of an operation with its "environment," in the Xfcc case study we used indices to check a property of this "environment." In more detail, the Chess contract describes that the result of GetMyGames identifies that GetMyGames's **receiver** service implements contracts Chess⟨(g['gameId'], moves(g), **False**)⟩ where gameId is the game ID of each game g in the result of the operation. Additionally, if a game's drawOffered flag is **True** (i.e., the opponent has offered a draw), the **receiver** service of GetMyGames also implements contract Chess⟨ (g['gameId'], moves(g), **True**)⟩. The following snippet puts these pieces together:

```
service Chess {
  GetMyGames(username, password)
  @foreach g in « result » identifies Chess at receiver with index « (g['gameId'], moves(g), False) »
  @foreach g in « result » identifies Chess at receiver with index « (g['gameId'], moves(g), True) »
    when « g['drawOffered'] == True »
  ...
}
```

The fact that a service implements Chess⟨(gameId,moveCount,True)⟩ indicates the existence of a draw offer for the game with ID gameId while the opposite indicates the absence of a draw offer. Moreover, the indices include the number of moves so far in a game, moves(g), as the "timestamp" of a draw offer. Thus indexed contracts give us a way to express and enforce draw offers: a client can accept a draw at a given time in a game (i.e., acceptDraw is **True** and the game has ID gameId and moveCount moves so far), only if the **receiver** service of MakeAMove implements Chess⟨(gameId,moveCount,True)⟩:

```
service Chess {
    ...
    MakeAMove(gameId, resign, acceptDraw, movecount, offerDraw, ...)
    @where index is « (gameId, movecount, acceptDraw) »
    @ensures « result != "NoDrawWasOffered" »
}
```

In the event that the player's opponent has not offered a draw for the game with their last move, the player attempts to accept a draw, and the post-condition of MakeAMove fails, Whip blames the client for deviating from the draw protocol.

7 PERFORMANCE

To evaluate how Whip impacts the performance of services it enhances, we analyze the time, memory, and network overhead due to Whip on the case studies from Section 6. We developed a test suite for each case study which exercises all the contracts from Section 6. All services are Whip-enhanced to maximize adapter traffic. Operations that identify a service entry always introduce the service entry (i.e., always use a new contract index and thus create new service entries in the adapters' local state, which maximizes local state size). We do not use the actual third-party services for our experiments but instead *mock* their behavior, i.e., we simulate their behavior with pre-computed responses for each request. This is for two reasons: (1) mocking services removes several sources of measurement noise, like service latency variation from background request load, and (2) performing our experiments on third-party production servers violates their terms of use.

We collect the following measurements for each test. First, we record the time to perform each request in the test suite and receive a reply for (1) the test client alone, and (2) the client enhanced with an adapter. The difference between these two measurements yields the latency due to the client's adapter per request (adapter latency). Second, we record the amount of memory (RAM and hard-disk) used by the client's adapter. Finally, we measure the adapter-to-adapter traffic (not including the original request or reply) in the TCP stream. We measure only the client's adapter as it is the hub for all communication in each experiment.

We ran our experiments on a 3 GHz Intel Core i7 processor with 2 GB of DDR3 memory with loopback communication. Figure 12 shows the experimental results for each test suite. Average adapter latencies for Twitter, Chess, and Evernote are 22ms, 55ms, and 59ms respectively. To place these measurements in context, the production versions of the case studies' services have latencies

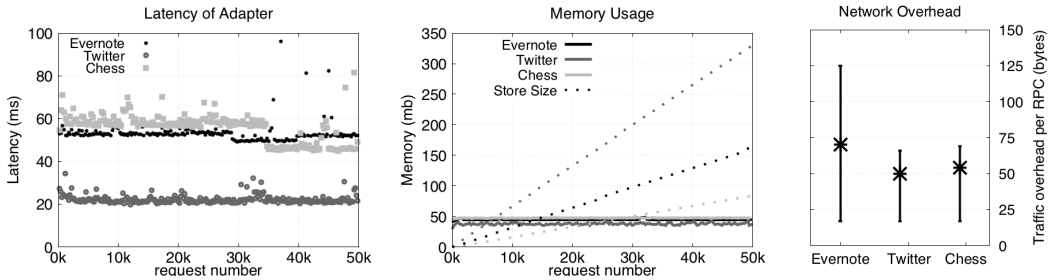

Fig. 12. The charts show the time, memory, and network overhead for each case study. The left chart shows the latency of the adapter as the number of requests increases. Each point is the average of the 250 requests around it. The middle chart shows the resident set size of the adapter and the dashed lines show the sizes of the store on disk. The right chart shows the average amount of adapter traffic per operation call. Vertical bars indicate 95% confidence intervals.

approximately 20 times greater than the average adapter latency.[17] The rate of increase in the disk-backed store for Twitter, Chess, and Evernote is 6.8kb per request, 1.7kb per request, and 3.3kb per request respectively. While disk-backed storage will increase without limit, the size of the in-memory cache is capped. The experimental results show that a cache size of just a few hundred megabytes suffices to cache adapter information for tens of thousands of requests. Average network overhead for Twitter, Chess, and Evernote is 50 bytes, 54 bytes, and 70 bytes per request respectively. Network overhead variance is from the operations identifying different service entries.

The network overhead and the rate of increase in store size depend on how many services each contract identifies. However, neither latency nor memory usage degrade as the number of requests increases despite an increasing network overhead and store size. Moreover, network overhead and store size do not have a dominant effect on latency; Twitter has the largest store and highest average network overhead yet the lowest latency. Instead, latency depends largely on the efficiency of the network plugin; the REST plugin uses a more efficient marshaller and handles sockets more performantly than the other plugins. Finally, all services in the experiments have a definite finite scope (according to their documentation) and so could be safely garbage collected at some point as discussed in Section 6.

8 RELATED WORK

Existing frameworks for composing services can enforce higher-order behavioral contracts similar to Whip's but assume that services are written and deployed in a particular manner. For example, CORBA [Object Management Group 2012], BPEL [Juric 2006], and Java RMI [Waldo 1998] require all services to use their libraries. Whip supports compositions of services that do not or only partly use these middlewares with appropriate message format plugins.

Behavioral Interface Specification Languages (BISLs), such as JML [Leavens et al. 2006], have extensions for specifying and enforcing higher-order behavioral contracts for communicating components. However, these languages are tightly coupled with particular component-implementation languages or families of languages. For instance, JML is designed for Java programs and has specific features to handle inheritance. Also, tools based on these languages re-write programs to insert checking probes. Thus BISLs and their contract checking tools are not language-agnostic. In contrast, Whip and its IDL are language-agnostic and do not modify services' code. Some features of Whip's IDL, such as pre- and post-conditions, are common with most BISLs. Others, such as **identifies**, are unique to Whip. Runtime verification tools, such as Monitor-Oriented Programming (MOP) [Chen and Roşu 2007], can in principle enforce higher-order behavioral contracts for

[17] For example, see https://dev.twitter.com/overview/status.

communicating components in a language agnostic manner (with appropriate plugins). However, building a contract system on top of them requires first solving the semantic issues Whip solves.

Many techniques exist to enhance the reliability of distributed systems (e.g., [Howard et al. 1988; Lamport 1998; Oki and Liskov 1988]) and are compatible with and orthogonal to Whip. Indeed, modern services are often chosen for organizational concerns such as loose coupling and scalability, rather than reliability. We focus on functional correctness of modern service composition. We briefly discuss how Whip affects the failure model of distributed applications in Section 3.

A wide range of frameworks enforce synchronization protocols of communicating components. For example, finite state machines can constrain the order of WSDL-defined interactions [Li et al. 2006] and web browsing [Hallé et al. 2010] in a manner complementary to that of Whip. BPEL [Juric 2006] is an expressive specification language for the orchestration of web services. Enforcement of BPEL, though, relies on a centralized communication bus for all services in an application. Multi-party session types [Honda et al. 2008] assume a global coordination protocol that is broken into locally and statically enforceable pieces. Further extensions marry multi-party session types with Design by Contract [Bocchi et al. 2010]. In general, dynamic monitoring of multi-party session types shares the same motivation as Whip [Hu et al. 2013]. Even though, in principle at least, the combination of session types with contracts and dynamic monitoring leads to specifications that subsume those of Whip, runtime verification of session types depends on annotating the source code of or using particular libraries by all components involved in a protocol. In contrast, the black-box treatment of (legacy) services and partial deployment are key aspects of Whip.

Closer to Whip, the work of Jia et al. [2016] describes the theory of a runtime monitor with precise blame for higher-order session types. Besides the fact that higher-order session types alone do not subsume Whip's higher-order contracts, there are important differences between the mechanism of Jia et al. and Whip. First, for correct blame, many operations of Jia et al.'s model (e.g., cut and forwarding) affect the topology of communication introducing indirect message queues. In contrast, Whip only affects the communication topology locally to each service and introduces no intermediate indirection to service communication. Second, for precise blame, the mechanism of Jia et al. requires that monitors have access to shared state. Whip adapters have access only to local state. Furthermore, Whip adapters do not need to exchange any extra messages to keep their local states in sync; new information is inferred from or piggybacked onto messages that services exchange. Third, it is unclear how their mechanism handles legacy services. For example, the use of explicit direction shift messages due to polarized session types is incompatible with existing message protocols. Whip is compatible with any protocol built on top of TCP.

9 CONCLUSION

Whip enhances modern services with higher-order behavioral contracts to bridge the semantic gap between simple network protocols and the higher-order properties of services. Whip comes with a higher-order contract language tailored to the needs of modern services. Moreover, Whip is transparent, suitable for partial deployment, and compatible with popular message formats. Thus, Whip promotes the correct composition of modern service-oriented applications, including legacy services, and with correct blame assignment facilitates their debugging and maintenance.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1421770. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. 2010. A Theory of Design-by-contract for Distributed Multiparty Interactions. In *Proceedings of the 21st International Conference on Concurrency Theory*. 162–176.

Feng Chen and Grigore Roşu. 2007. MOP: An Efficient and Generic Runtime Verification Framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. 569–588.

Christos Dimoulas, Robert Bruce Findler, Cormac Flanagan, and Matthias Felleisen. 2011. Correct Blame for Contracts: No More Scapegoating. In *Proc. 38th SOSP*. 215–226.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*. 48–59.

Martin Fowler and James Lewis. 2014. Microservices. (2014). http://martinfowler.com/articles/microservices.html

Sylvain Hallé, Taylor Ettema, Chris Bunch, and Tevfik Bultan. 2010. Eliminating Navigation Errors in Web Applications via Model Checking and Runtime Enforcement of Navigation State Machines. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering*. 235–244.

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 273–284.

John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. 1988. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 51–81.

Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda. 2013. Practical Interruptible Conversations - Distributed Dynamic Verification with Session Types and Python. In *Runtime Verification - 4th International Conference*. 130–148.

Limin Jia, Hannah Gommerstadt, and Frank Pfenning. 2016. Monitors and blame assignment for higher-order session types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 582–594.

Matjaz B. Juric. 2006. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. Packt Publishing.

Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169.

Gary T Leavens, Albert L Baker, and Clyde Ruby. 2006. Preliminary design of JML: A behavioral interface specification language for Java. *Software Engineering Notes* 31, 3 (2006), 1–38.

Zheng Li, Yan Jin, and J. Han. 2006. A runtime monitoring and validation framework for Web service interactions. In *Proceedings of the Australian Software Engineering Conference*. 10–79.

Bertrand Meyer. 1988. *Object-oriented Software Construction*. Prentice Hall.

Bertrand Meyer. 1991. Design by Contract. In *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1–50.

Bertrand Meyer. 1992. Applying Design by Contract. *IEEE Computer* 25, 10 (1992), 40–51.

Object Management Group. 2012. *CORBA Component Model*. Specification Version 3.3. http://www.omg.org/spec/CORBA/3.3/

Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. ACM, New York, NY, USA, 8–17.

Randy Shoup. 2015. Service Architecture at Scale: Lessons from Google and eBay. (2015).

Jim Waldo. 1998. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency* 6, 3 (1998), 5–7.