Building Systems of Systems with Escher

Burcu Canakci(✉), Lorenzo Alvisi, and Robbert van Renesse

Cornell University, Ithaca, USA
{burcu,lorenzo,rvr}@cs.cornell.edu

Abstract. This paper presents Escher, an approach to build and deploy multi-tiered cloud-based applications, and outlines the framework that supports it. Escher is designed to allow *systems of systems* to be derived methodically and to evolve over time, in a modular way. To this end, Escher includes (i) a novel authenticated message bus that hides from one another the low-level implementation details of different tiers of a distributed system; and (ii) general purpose wrappers that take an implementation of an application and deploy, for example, a sharded or replicated version of an application automatically.

Keywords: Middleware · Distributed systems · Refinement · Fault tolerance · Maintainability

1 Introduction

Large-scale distributed systems deployed today within and across datacenters are hierarchically structured. At the highest level they are decomposed into *tiers*, such as, for example, the load balancing tier, the web frontend tier, the data store tier, the caching tier, the data analysis tier, the recommendation tier, and various application logic tiers. Each tier in turn is further subdivided into smaller components, down to the executable services that run on the datacenter servers. Such a *refinement hierarchy* allows for modular development and simplifies management. Furthermore, at the lowest level applications are often containerized to simplify software distribution and deployment. Various readily available management services maintain configurations, dependencies, and deployments of such systems [7,10,22].

While this approach has produced distributed systems of unprecedented scale and sophistication, these systems of systems have introduced a new challenge: managing the interaction between tiers in this refinement hierarchy. Standards for data representation and remote invocation have made it straightforward to glue together services written in different programming languages or deployed on different operating systems [12]. However, such ease of integration stops at the implementation level in the refinement hierarchy; still lacking is a way to describe abstractly the interaction between higher-level tiers of the system.

C. Johnen et al. (Eds.): SSS 2021, LNCS 13046, pp. 34–50, 2021.
https://doi.org/10.1007/978-3-030-91081-5_3

Consider the architecture of a search engine. A client's query first goes through a query processor which is responsible for parsing and formatting the query. The request is then relayed to an online ranking tier, which computes the results. That tier interacts with an indexing tier which is responsible for crawling and indexing offline. The indexing tier itself can include different storage systems, such as an in-memory caching system for popular queries and a disk storage database system for historical indexes. Many modern applications are similarly composed of numerous interacting tiers and components [2,6,24,28]. Different components have different non-functional objectives, and so require different refinements. Additionally, these objectives may change over time, so their refinements are not static [24]. Therefore, what is needed is a clean abstraction for the interaction between *any* two refined components, one that allows them to independently evolve, or even be replaced, without affecting the rest of the system.

This paper presents *Escher*, a principled methodology to (*i*) systematically derive such systems of systems and (*ii*) continue supporting their organic evolution.

Escher specifies communication between tiers at the highest levels of their refinement hierarchy. Instead of exposing a plethora of ways for components to interact with one another, Escher offers just one: a novel message bus that is *refinement-aware*, in that it hides the low-level, refinement-specific details of how components interface with one another. For example, it transparently manages the interaction between a replicated data store and a sharded caching service, and it automatically handles how one replicated component interacts with another in a fault-tolerant fashion.

Escher's bus exposes only the *external* behavior of a system's components. Enforcing this form of isolation is critical to supporting the organic growth of the system: it allows the implementation of each component to be refined (and its correctness to be verified) modularly, and thus for it to evolve, or even be replaced, without requiring other components that are in communication with it to also be reconfigured, or recompiled with new communication libraries.

To further facilitate organic growth, Escher transparently supports modular transformations with *wrapper*s, which can automatically provide components with desirable properties such as fault tolerance, load balancing, or privacy.

Escher goes well beyond conventional message streaming platforms [16,23], middleware services [14,20,22], or service-oriented-architecture solutions, such as enterprise service buses [12]. While these solutions enable interactions between objects or services implemented using different languages, data models, or communication APIs, they provide isolation only at the implementation level. Further, their ability to support a system's organic growth is limited. For example, CORBA [14] can refine a service to add fault-tolerance through replication, but does not support the same capability for clients. In Escher, all tiers can be fully and independently refined. For example, a replicated client can transparently communicate with a replicated server, regardless of the replication protocol

used on either side. To our knowledge, Escher is the first system to support such two-sided transparency and generality for any refinement.

This paper covers in detail the system model underlying Escher's refinement hierarchy in Sect. 2 and presents different implementation options in Sect. 3. We discuss related work in Sect. 4.

2 Escher Design

Escher models components as *agents*; whether implemented by a single process or a collection of processes, they consist of a state machine combined with a message inbox and outbox. The Escher message bus moves messages from outboxes to inboxes. An agent can be refined by replacing it with a non-empty set of new agents, each with their own state machines, inboxes, and outboxes. A refinement mapping [1] specifies how inboxes and outboxes of the new agents map to the inbox and outbox of the original agent. Escher uses this mapping to hide refinement from other, unrelated agents.

Agents form a *refinement hierarchy*: a parent agent is refined into a set of child agents. For example, in a replicated service, the parent is a virtual, logically centralized service, while the children are the replicas that together provide that abstraction. Other refinements can be similarly modeled and applied recursively to render the refinement hierarchy.

Each agent in the refinement hierarchy has a separate management interface. In particular, each agent has its own public/private key pair. An agent that is being refined uses its private key to create public key certificates for its child agents, so they can prove that they are part of a specific refined agent. The root of the refinement hierarchy forms a *Certification Authority* (CA) for the entire system. The leaves in the hierarchy are physical agents: the Escher framework uses their private keys to sign and verify their messages.

The Escher refinement hierarchy thus forms a uniform management interface for all agents, whether virtual (refined) or physical (running code). Escher therefore solves not only the interfacing between agents, but also provides the management necessary to successfully deploy and run a system of distributed services.

Escher provides the following design properties (partially borrowed from [11]):

Transparency: The refinement of an agent is transparent to other agents. An agent only requires the high-level API (through a set of message types) and the identifier of a destination agent to communicate with it, even if the destination is refined.
Generality: Agents can be refined arbitrarily, as long as the refinement is correct.
Flexibility: Agent refinements can evolve over time.
Manageability: Agents, whether refined or not, can be deployed, executed, and monitored uniformly.

2.1 System Model: Agents and Message Bus

An Escher system is a collection of agents communicating over a message bus. Each agent a is identified by a unique identifier, denoted $a.\texttt{id}$. Each agent a includes a (possibly nondeterministic) state machine and three unordered collections of messages: $a.\texttt{inbox}$, $a.\texttt{outbox}$, and $a.\texttt{donebox}$. The inbox and done box of an agent are initially empty. The operation of a correct agent a proceeds as follows:

1. Wait until there exists a message $m \in a.\texttt{inbox} - a.\texttt{donebox}$;
2. Apply m to the state machine to produce a new state and a set of output messages;
3. Add the output messages to $a.\texttt{outbox}$;
4. Add m to $a.\texttt{donebox}$;
5. Repeat.

A message is a four-tuple: ⟨message identifier, source agent identifier, destination agent identifier, payload⟩. No two messages from the same correct source agent for and the same destination agent can have the same message identifier.

The operation of the message bus is as follows:

1. Wait until there exists a message $m = \langle \_,\ s.\texttt{id},\ d.\texttt{id},\ \_ \rangle$ such that $m \in s.\texttt{outbox} - d.\texttt{inbox}$ ($\_$ denotes a wildcard);
2. Add m to $d.\texttt{inbox}$;
3. Repeat.

Logically, the contents of the three boxes of a correct agent grow monotonically. Later, we show how messages that have been handled can be garbage collected.

The system is asynchronous but fair: if a correct agent—or the message bus—can continually take a step, it eventually will. Unless refined to be fault-tolerant, agents can exhibit both crash failures and arbitrary (Byzantine) failures:

- An agent a that experiences a crash failure no longer processes messages in $a.\texttt{inbox} - a.\texttt{donebox}$;
- An agent a that experiences a Byzantine failure ignores its state machine and places any messages in $a.\texttt{outbox}$. It may also remove messages from $a.\texttt{outbox}$.

The message bus is reliable and guarantees *authenticity*: if agents s and d are correct and message $m = \langle i,\ s.\texttt{id},\ d.\texttt{id},\ p \rangle \in d.\texttt{inbox}$, then $m \in s.\texttt{outbox}$.

2.2 Refining Agents

An agent a can be refined, that is, replaced with one or more new agents $a_1, ..., a_n$ such that $a_1, ..., a_n$ collectively have the same *external behavior* as a. We call a the *parent agent* of $a_1, ..., a_n$ and $a_1, ..., a_n$ the *child agents* of agent a. We denote C_a to be the set of child agents of agent a. Child agents can be refined as well, leading to a hierarchy of agents; at the root of the refinement hierarchy is a

single *system agent*, which models the entire system. Agents are then identified by *pathnames* of the form {/id}*. For example, /x/y identifies a child agent of /x. The system agent is identified by the empty path name "".

Given a set of input messages M^I for agent a, agents $a_1, ...a_n$ must produce the same set of output messages M^O that agent a might have; specifically, whether or not a correct agent a has been refined must be *invisible* to other correct agents that a communicates with. Supporting this notion of indistinguishability requires adding important features to the message bus.

First, it introduces restrictions on who can communicate with whom. A correct agent will only send messages to its siblings or siblings of its ancestors. For example, a correct agent identified by /x/y/z may send messages to agents identified by /x/y/z′, /x/y′, or /x′, but not to agents identified by /x/y/z′/w, /x/y′/z′ or /x′/y′.

Second, it requires the message bus to be refinement-aware. In particular, if an agent d is refined, the message bus attempts to deliver a message $m = \langle i,\ s.\texttt{id},\ d.\texttt{id},\ p\rangle$ from the outbox of a correct agent s to the inboxes of all child agents of agent d. The message bus guarantees that each message in the inbox of a correct agent d has a destination agent identifier that is a prefix of the identifier of d. For example, the message bus may deliver a message destined for x/y to agent x/y/z, but not a message destined for x/y/z/w nor a message destined for x/y/z'.

In a refinement of agent a, there must exist a function R_a that maps the state of the child agents to the state of the parent agent a [1]. Because the state of each agent has four components (inbox; outbox; done box; and state of its state machine), we split R_a into four corresponding components R_a^I, R_a^O, R_a^D, and R_a^S. In particular, to be able to exchange messages between refined agents, the message bus needs to understand two of these components: R_a^I and R_a^O.

R_d^I is an application-dependent function that, applied to the inboxes of d's the child agents, computes $d.\texttt{inbox}$. For example, in the case of replication, a message could be considered in $d.\texttt{inbox}$ once it is delivered to the inbox of a correct child agent of d; In the case of sharding, a message is considered delivered once it is delivered to the inbox of the child agent that can handle the request in the message.

Similarly, R_s^O is an application-dependent function that, applied to the outboxes of s's child agents, computes $s.\texttt{outbox}$. Let μ_s map a set of messages in the outboxes of child agents of s to a set of messages in the outbox of s. Formally, μ_s maps a set of messages $M_{\langle i,s,d\rangle}$ of the form $\langle i,\ c.\texttt{id},\ d.\texttt{id},\ p_c\rangle$, $c \in C_s$, to a set of messages of the form $\langle i,\ s.\texttt{id},\ d.\texttt{id},\ p_s\rangle$. Recall that we require that correct agents do not produce more than one message with the same message identifier, source agent identifier, and destination agent identifier. Therefore, the result of μ_s is either the empty set or a singleton set. Moreover, if $\mu_s(M^1_{\langle i,s,d\rangle}) = \{\langle i,\ s.\texttt{id},\ d.\texttt{id},\ p_s^1\rangle\}$ and $\mu_s(M^2_{\langle i,s,d\rangle}) = \{\langle i,\ s.\texttt{id},\ d.\texttt{id},\ p_s^2\rangle\}$, then $p_s^1 = p_s^2$: messages from different subsets of child agents with the same message identifier must map to the same payload.

For example, if the refinement is "replication for crash failures", then μ_s is the identity function: a message from any of s's child agents (replicas) is a message from the parent agent s. In crash tolerant replication all replicas generate the same messages; thus, applying μ_s to non-empty sets produces singleton sets. Moreover, μ_s applied to any non-empty $M_{\langle i,s,d\rangle}$ is guaranteed to produce the *same* singleton set.

However, if the refinement is "replication for Byzantine failures" with some parameter f, then

$$\mu_s(M_{\langle i,s,d\rangle}) = \left\{ \langle i,\ s.\texttt{id},\ d.\texttt{id},\ p\rangle \;\middle|\; c \in C_s \ \wedge \left| \bigcup \{ \langle i,\ c.\texttt{id},\ d.\texttt{id},\ p\rangle \in M_{\langle i,s,d\rangle} \} \right| > f \right\}$$

That is, a message from s must have more than f matching messages from its child agents. Again, for Byzantine replication, the result of μ_s is either the empty set or some singleton set containing some message m. If $|M_{\langle i,s,d\rangle}| \leq f$, then the result is guaranteed to be the empty set; if $|M_{\langle i,s,d\rangle}| \geq |C_s| - f$, then the result is guaranteed to be $\{m\}$.

Let $M_{\langle i,s,d\rangle}$ be a set of messages such that $\mu_s(M_{\langle i,s,d\rangle})$ yields a singleton set containing a message from s. Then, we call $M_{\langle i,s,d\rangle}$ a *merge group* of messages. Every message in a merge group has an identical message identifier, destination agent identifier, and parent of the source agent.

Let $s.\texttt{outbox} = R_s^O(\{c.\texttt{outbox} \mid c \in C_s\})$ be the set of messages in the outbox of agent s according to the refinement mapping of s. We require that the μ_s function satisfies

- **Soundness**:

$$\forall M_{\langle i,s,d\rangle} \ \subseteq \{\langle i,\ c.\texttt{id},\ d.\texttt{id},\ p_c\rangle \in c.\texttt{outbox} \mid c \in C_s\} : \\ \mu_s(M_{\langle i,s,d\rangle}) \subseteq s.\texttt{outbox}$$

  In other words: all messages from s that can be constructed, using μ_s, from the outboxes of the child agents of s, are in fact messages produced by s.
- **Completeness**: If $m = \langle i,\ s.\texttt{id},\ d.\texttt{id},\ p\rangle \in s.\texttt{outbox}$, then eventually there exists a set $M_{\langle i,s,d\rangle}$ from correct agents in C_s such that $m = \mu_s(M)$. In other words, every message in $s.\texttt{outbox}$ is eventually retrievable from the correct child agents of s.

As mentioned above, the message bus delivers a message sent to a refined agent d to the inboxes of all its child agents. This is sufficient, but depending on the definition of R_d^I may be overkill. Escher allows agents to specify a minimum refinement "depth" to manage delivery more precisely. The minimum depth is the number of components in the destination path of the message. For example, an agent with identifier `/x/y/z` and a minimum depth of 2 would only receive messages destined to `/x/y/z` and `/x/y`, but not to `/x`. The default minimum depth is 1.

To summarize, the message bus collects messages from the child agents of s with the same message identifier and destination agent identifier, and uses

μ_s to "merge" those messages into messages from s. If the destination of such a message m is d, then the message bus delivers m to d.`inbox`. If agent d is also refined, then the message bus delivers m to the inboxes of d's child agents, constrained by the child agents' minimum depth settings.

2.3 Wrappers

In some common refinements, which include state machine replication and sharding, a child agent's state machine uses the state machine transition function of its parent as a subroutine. Escher includes a special type of agent called a *wrapper* to handle these types of refinements automatically.

For example, in the case of state machine replication, the wrappers are the replicas, and they must agree on the order in which they present incoming commands to their copies of their parent's state machine (which is required to be deterministic). Similarly, in the case of sharding, each wrapper is responsible for a range of keys and has a copy of the parent's state machine transition function. Each wrapper passes messages for keys within the shard's range to its copy of the state machine.

Escher contains a collection of general-purpose wrappers that automatically provide certain desirable *non-functional properties* to agents such as being able to survive certain classes of failures or being able to handle high load.

2.4 Deploying and Managing Agents

We separate agents into *physical agents* and *virtual agents*. Physical agents are not refined and form the leaf agents in the refinement hierarchy. Virtual agents are refined—they are the internal nodes in the refinement hierarchy. Each agent, whether physical or virtual, must be *managed*. If a is a virtual agent, its manager mgr_a is a principal that can be a person, a cluster manager, or even another agent; if a is a physical agent, mgr_a is a *runtime* that runs both the code of the leaf agent and Escher code that implements the functionality of the message bus.

Escher includes a Public Key Infrastructure (PKI) that it uses to prevent Byzantine agents from forging messages from correct agents: in particular, it ensures that if agent s is correct, then messages that claim to come from s are guaranteed to be in s.`outbox`. The PKI associates with each agent, whether physical or virtual, a private key p_a and a corresponding public key certificate P_a. For a correct agent, only mgr_a holds p_a; the private keys and public key certificates are generally transparent to the agents themselves.

The public key certificate P_a of agent a binds its identifier (a path) to its public key and is signed using the private key of the parent agent of a. Therefore, the manager of a virtual agent a is a Certification Authority (CA) to the child agents of a. The system agent is the root CA of Escher's PKI. The public key certificate of the system agent is self-signed.

Public key certificates are *chained*: they refer to the public key certificates of their parents. Thus, to refine an agent a, manager mgr_a uses its private key p_a

to generate a public key certificate for each of the child agents. Afterward, mgr_a can be offline, unless the refinement of a is updated. If the identifier of some agent x is $\langle \texttt{path} \rangle \texttt{/x}$, then the next certificate on the certificate chain of x is the public key certificate of the parent of x, identified by $\langle \texttt{path} \rangle$.

If a is a physical agent, the runtime that manages a, among other functions described in the implementation section below, signs outgoing messages using p_a and attaches to them the certificate chain (although certificates can be cached for efficiency). The runtime of destination agents can then verify the messages using the public key contained in P_a.

2.5 Compatibility with Legacy Services

To enable its own gradual deployment, Escher can incorporate legacy services.

Existing centralized microservices simply require a *shim*, which is associated with an Escher identifier and implements the inbox, outbox, and done box. The shim must define a set of message types through which other Escher agents can communicate with it. Upon receipts of such a message, the shim has to invoke the corresponding functionality in the microservice. The output of the microservice has to be similarly transformed before it can be sent on the Escher bus.

Legacy distributed services can be incorporated into Escher in a similar fashion, by defining a virtual agent that models the distributed service. The actual running components of the service can then be modeled as its child agents, each paired with its own shim.

3 Implementation

In this section, we describe several implementation options for the Escher message bus as well as the choices we favor.

3.1 Message Merging

The Escher *runtime* of a physical agent a maintains the inbox, outbox, and done box for a, as well as the state of its state machine. Each runtime also partially implements the message bus. An important functionality of the message bus is to collect messages from the child agents of a refined agent, and to merge them into the corresponding message from the parent agent before delivering the message to the destination. But which component(s) of an Escher system should be responsible for merging messages?

To frame the question, let s be a refined agent and $m_s = \langle i,\ s.\texttt{id},\ d.\texttt{id},\ p_s \rangle$ a message from s to d. Assume child agents of s are unrefined for now. To merge messages from the child agents of s into m_s, an Escher component needs to: (1) collect the necessary set $M_{\langle i,s,d \rangle}$ of messages from the child agents of s, and (2) apply a merge function μ_s to $M_{\langle i,s,d \rangle}$.

Merging at the Sender. One straightforward candidate to perform the merging would be the runtime of one of the child agents of s. This runtime would be in charge of merging outgoing messages and of sending merged messages to agents external to the refinement. After all, the runtime of a child agent of s already knows μ_s, and it can collect $M_{\langle i,s,d\rangle}$ from the runtimes of other child agents. However, this solution comes with several drawbacks: (*i*) it is not fault tolerant; (*ii*) it requires additional cryptographic support, such as multisignatures or threshold signatures, to provide message authenticity efficiently; and finally (*iii*) it requires a message from s to d to take two steps of communication: one from the runtimes of other child agents to the "merging" runtime, and one from the merging runtime to the runtime(s) of the physical descendants of d.

Merging in the Middle. A second solution would be to have a logically-centralized trusted Escher service handle all message merging in the system. This "merging" service, however, would need to be replicated for fault-tolerance, and may become a bottleneck for the entire system. Further, with this solution, as with the previous one, a message from s to d would take at least 2 steps of communication.

Merging at the Receiver. Escher opts for a third solution: it leaves the runtime of a physical destination agent d' the responsibility of merging the messages it receives. To merge messages from child agents of a refined agent s, d''s runtime needs to know μ_s. We considered endowing Escher with a trusted configuration service, which d' could query to retrieve μ_s, but we ultimately rejected this option: the service would add an extra step of communication before the runtime of d' can merge messages, would have to be replicated for fault tolerance, and may turn into a bottleneck. Further, child agents in dynamic refinements would need to synchronize their configuration with the service whenever the refinement is updated. Instead, we opted for an approach that eliminates the need for a global service and does not induce extra steps of communication: we piggyback μ_s onto all message from the child agents of s to agents that are not part of the refinement of s. When the runtime of a child agent attaches the corresponding merge function(s) to an outgoing message, we say that the runtime *tag*s the message.

This solution has minimal communication overhead, is fault tolerant, and both modular and general. Its main drawback is that running merging code in the destination agent is a potential security issue; thus, it should be done with great care, for example through the use of a safe language.

Implementing Merge Functions. Escher is intended to support any kind of refinement, and therefore specific merge functions should ideally not be built into the Escher framework itself. The most general option would be to support arbitrary programs for merge functions. However, a Byzantine agent can be refined as well, and such a Byzantine agent could issue certificates that verify a potentially

malicious program μ_F. The runtime of a correct agent that received a message from a child of the Byzantine agent could be damaged if it attempted to run μ_F without any sandboxing mechanism. An obvious alternative would be for Escher to ask programmers to assemble merge functions using a safe domain-specific language (DSL), such as SQL (modelling messages as rows in a relational table and using SQL aggregation functions to merge them), or the *extended Berkeley Packet Filter* [27], which, though a verifier, ensures that an eBPF program runs only for a limited time, accesses only a restricted memory region, and calls only an allowed set of safe external functions. Message processing abstractions from specification languages for distributed protocols, such as [19], can also be used in creating a DSL for merge functions.

We leave exploring such options for future work. Escher's current merge functionality can support a variety of common quorum-based crash- and Byzantine-tolerant replication protocols, as well as sharding protocols. The functionality is implemented as a dictionary of keys to values. A key defines a specific action that a runtime should take in order to merge a message; and the value customizes this action. We currently support two such pairs. The first is $(\texttt{wait for matching}, k)$, which informs the receiving runtime to wait until there are k messages with matching payloads. The second is $(\texttt{from} : a_1.\texttt{id}, ..., a_n.\texttt{id})$, which informs the runtime to only consider messages that are from a specified set of agents. For example, the merge function of a crash-tolerant replicated service with replicas $r_1, ..., r_n$ can be implemented with $\{\texttt{wait for matching} : 1,\ \texttt{from} : r_1.\texttt{id}, ..., r_n.\texttt{id}\}$ whereas the merge function of a Byzantine fault-tolerant replicated service can be implemented with $\{\texttt{wait for matching} : f + 1,\ \texttt{from} : r_1.\texttt{id}, ..., r_n.\texttt{id}\}$.

3.2 Message Delivery

The second functionality of the message bus is to deliver a message, addressed to a destination agent d that is refined, to the inboxes of all interested child agents of d. This functionality could be implemented using point-to-point communication between runtimes. The runtime of the sender, however, would need to know the addresses of d's child agents, which is difficult because refinements may be dynamic and addresses may change. Another option would be to have the runtime of a specific child agent of d receive all incoming traffic, acting as an ingress proxy. The child agent could then forward each message to their appropriate destinations. Unfortunately, while simple, this is not a fault-tolerant solution.

In Escher, runtimes discover one another using a publish/subscribe messaging service. This service has the following properties:

1. Runtimes *publish* messages to *topic*s. A topic is an agent identifier, either virtual or physical.
2. The runtime of an agent s can publish a message $m = \langle \_,\ s.\texttt{id},\ d.\texttt{id},\ \_ \rangle$ in $s.\texttt{outbox}$ to the topic named $d.\texttt{id}$.
3. The runtime of an agent s automatically *subscribe*s to the topic named $s.\texttt{id}$; it also subscribes to prefixes of $s.\texttt{id}$ according to its minimum depth (see Sect. 2).

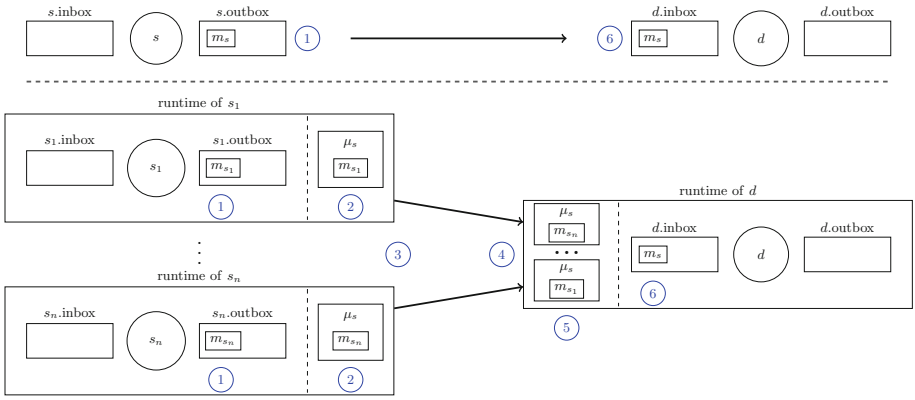

Fig. 1. Above the dashed line is a virtual representation of communication between agents s and d. (1) Virtual agent s sends a message $m_s = \langle i,\ s.\texttt{id},\ d.\texttt{id},\ p_s \rangle$ to d. This maps to a set of messages of the form $m_{s_i} = \langle i,\ s_i.\texttt{id},\ d.\texttt{id},\ p_{s_i} \rangle$ in the outboxes of the physical child agents of s. (2) The runtimes of the child agents of s tag the messages m_{s_i} with μ_s. (3) The runtimes of child agents of s send tagged messages to topic $d.\texttt{id}$. (4) The runtime of d receives tagged messages. (5) The runtime of d merges these messages into m_s. (6) Finally, the runtime of d places m_s in $d.\texttt{inbox}$.

Runtimes may opt to use the publish/subscribe messaging service for all communication. While attractive in its simplicity, this approach can negatively impact performance, as communication through pub/sub services typically results in significantly higher latencies than using point-to-point communication. Thus, the current Escher runtimes use the publish/subscribe service only to discover other agents; all ensuing communication is point-to-point. Should refinements change after this initial handshake, the runtimes need to repeat the process of discovery and connection.

3.3 Implementing Tagging and Merging

We describe now in detail (and illustrate in Fig. 1) the tagging and merging process that allows two agents s and d to communicate.

Let s be an agent refined into n child agents $C_s = \{s_1, ..., s_n\}$. Let $m_s = \langle i,\ s.\texttt{id},\ d.\texttt{id},\ p_s \rangle$ be a message in $s.\texttt{outbox}$ to be delivered to $d.\texttt{inbox}$. For example, s could be a replicated server sending a response to a client agent d. For simplicity, assume that d is physical (the merge functionality is the same for virtual destinations). Because refinement is transparent in Escher, m_s must be constructed by the message bus from messages that are physically in the outboxes of the child agents of s. The completeness property of μ_s implies that there exists a merge group $M_{\langle i,s,d \rangle}$ of messages of the form $m_{s_i} = \langle i,\ s_i.\texttt{id},\ d.\texttt{id},\ p_{s_i} \rangle$ in the outboxes of correct agents in C_s such that $\mu_s(M_{\langle i,s,d \rangle}) = m_s$. The runtimes of these child agents tag their messages with μ_s and send them to the topic $d.\texttt{id}$.

Note that messages between sibling agents (say replicas of the same server) do not need to be tagged.

The runtime of d eventually receives all messages sent by correct agents to $d.\texttt{id}$ and groups together messages with the same tag. Let M be a set of messages with message identifier i, destination agent identifier $d.\texttt{id}$, and source agent identifier a child of s. The runtime of d updates M whenever a new message with matching values is received, and applies μ_s to M after each update. If this results in the empty set, the runtime waits for additional messages. If, however, the result is a non-empty set, then it is guaranteed to be a singleton set containing m_s; in this case, the runtime delivers m_s in $d.\texttt{inbox}$ and discards M.

Generalizing for Multi-step Refinements. Let a be an agent that has $k+1$ ancestors with $k \geq 0$. Then, a is associated with a list of functions $\mathscr{M}_a = [\mu_k, ..., \mu_0]$, where μ_k is the merge function of a's parent and μ_0 is the merge function of the system agent. (μ_0 is not explicitly defined and never invoked.) We call $\mathscr{M}_a$ the *μ-list* of a.

If a is a correct agent with $k+1$ ancestors, its runtime may tag an outgoing message with a list of 0 to k tags, depending on the destination agent identifier. Given a message $\langle \_,\ a.\texttt{id},\ d.\texttt{id},\ \_ \rangle$ in $a.\texttt{outbox}$, the runtime determines the longest common prefix of $a.\texttt{id}$ and $d.\texttt{id}$. Suppose the longest common prefix has length l, $0 \leq l \leq k$. The runtime tags the message with the first $k - l$ functions in the μ-list of a.

As an example, consider an agent `/a/1/1`. The runtime of `/a/1/1` has the μ-list $[\mu_2, \mu_1, \mu_0]$. μ_2 is used to construct messages from virtual agent `/a/1` and μ_1 is used to construct messages from virtual agent `/a`. Messages from `/a/1/1` to `/a/1/2` need not be tagged, since these are sibling agents. A message from `/a/1/1` to `/a/2` needs to be tagged only with $[\mu_2]$, since `/a/2` and `/a/1` are sibling agents, and the refinement of its sibling is transparent to `/a/2`. A message from `/a/1/1` to `/d` is tagged with $\mathscr{M}_a = [\mu_2, \mu_1]$; it first needs to be merged using μ_2 to obtain a message from `/a/1` to `/d`, which in turn needs to be merged using μ_1 to obtain a message from `/a` to `/d`.

Each successful application of a merge function by the runtime removes a tag from a message. Runtimes can only place in their agent's inbox messages with no tags.

Since merge functions are provided by runtimes that may be faulty, the Escher PKI must account for tag authenticity; that is, runtimes need to verify tags. Without tag authenticity, the runtime of a faulty agent could tag its outgoing messages incorrectly. For instance, Byzantine replicas of a replicated service could collude and tag their messages with a faulty μ_F such that μ_F yields a singleton set containing a bad message. To prevent this, recall that a manager of a virtual agent s issues a public key certificate to its child agents by signing their public key and identifier. We require this certificate to also includes a hash of the merge function μ_s.

3.4 Garbage Collection of Delivered Messages

An agent's inbox and outbox are specified as append-only. In practice, however, once a message is in the done box of the destination agent, it can physically be removed from the corresponding in- and outboxes. In this section, we discuss how.

We are explicitly not concerned with garbage collecting messages in an agent's done box, as the done box is outside Escher's purview. Further, while at the specification level it is another unbounded message box, an application can often implement its functionality using, say, a sequence number per client to track which messages it has handled thus far.

However, to implement garbage collection of inboxes and outboxes, a runtime must be able to find out whether a message with a given identifier is in the done box. To this end, we require each agent to expose a method `contains`$(i,\ s.\texttt{id},\ d.\texttt{id})$ that returns `true` iff there exists a message $\langle i,\ s.\texttt{id},\ d.\texttt{id},\ \_\rangle \in a.\texttt{done}$.

Garbage Collection in Outboxes. A message $m = \langle i,\ s.\texttt{id},\ d.\texttt{id},\ \_\rangle$ can be removed from $s.\texttt{outbox}$ once it is in $d.\texttt{inbox}$. To implement garbage collection of agent outboxes, Escher uses acknowledgment messages. When the runtime of an agent d delivers a message $m = \langle i,\ s.\texttt{id},\ d.\texttt{id},\ \_\rangle$ to $d.\texttt{inbox}$, it sends an `ACK` message $m_{\texttt{ACK}} = \langle \texttt{ACK}_i,\ d.\texttt{id},\ s.\texttt{id},\ m\rangle$ to agent s. `ACK` messages must be tagged the same way as regular messages. The message identifier of an `ACK` message is of the form $\langle ACK,\ i\rangle$, where i is the message identifier of the message being acknowledged. When the acknowledgment message is received (possibly after merging) by s or one of its descendants, the runtime removes m from its outbox. `ACK` messages are exchanged between runtimes only and are ephemeral: they are not placed in inboxes or outboxes and are not themselves acknowledged.

Garbage Collection in Inboxes. A message $m = \langle i,\ s.\texttt{id},\ d.\texttt{id},\ \_\rangle \in a.\texttt{inbox}$ can be removed once it is in $a.\texttt{done}$. The runtime of an agent a periodically evaluates $a.\texttt{done.contains}(i,\ s.\texttt{id},\ d.\texttt{id})$ for each such message. Also, if an unmerged *fragment* of a message arrives, the `contains` method is invoked to see if the fragment must be stored for later merging. If the message is already in the done box, an acknowledgment is sent but the fragment is otherwise dropped to avoid duplicate delivery.

4 Related Work

Many frameworks have been proposed that simplify the development of distributed systems. Here we only cover the projects most related to Escher.

Object-Oriented Middlewares. A variety of middlewares propose modeling a distributed system as a collection of objects and provide uniform interfaces to manage and access those objects, including CORBA [14], JavaBeans [26], and DCOM [20]. For example, CORBA defines an Interface Definition Language to

specify object interfaces and allows applications to be written in a variety of programming languages and deployed on a variety of operating systems. An Object Request Broker allows objects to invoke interfaces of other objects transparently. To this end, both client and servers have *stubs* that automatically marshal arguments and results. Herein also lies the problem that Escher solves: these stubs do not support refinement. While CORBA does provide support for an unrefined client to securely interact with a refined server, it cannot transparently support multi-tiered distributed services in which refined clients interact with refined services. While Escher can support an object-oriented approach to application development, Escher is more general and supports other popular paradigms for distributed systems development such as streaming applications.

Reconfigurable Middlewares. Reconfigurable distributed systems support the replacement of their sub-systems. The aforementioned CORBA supports reconfiguration, see for example [8]. Ajmani et al. [3] supports multi-version systems and can gradually upgrade software of distributed systems. Escher supports agent upgrades at any level of refinement and therefore supports both upgrades of individual physical agents and upgrades of an entire distributed service.

Refinement-based Middlewares. The Ovid system [5] supports refinements (in the form of so-called *transformations*). The Escher refinement hierarchy is similar to Ovid's Logical Agent Transformation Tree, and Escher borrows the concept of using pathnames for refined agents from Ovid. However, Ovid does not support a uniform and secure way for refined agents to communicate, a key feature of Escher. The Ovid management system depends on a *controller agent*, which needs to be programmed with specific code for each type of agent and refinement and leaves open the question of who manages the control agent itself. The Escher management interface is uniform and does not require application- or refinement-specific code. Verdi [29] is another system that supports transformations. It is focused on formal verification of such transformations using a Coq-based toolchain. IronFleet [15] supports building and verifying distributed systems using the Dafny [18] language. Neither Verdi nor Ironfleet provide uniform interfaces for refined agents to communicate nor any form of application management.

The authors of [17] use stepwise refinement, to develop fault tolerant middleware for mobile agent systems. The middleware allows groups of agents to work together in isolated *scopes* and lets individual agents move around, switching scopes, while supporting interprocess communication via the Linda Tuple Space [13]. In [25], the authors describe a reflexive approach to updating communication between JavaBeans components in the face of reconfiguration events. While also based on refinement, neither approach solves communication between refined services, which is the key problem Escher addresses.

Replicated Remote Procedure Call. Perhaps the earliest work investigating how refined services can interact is *replicated procedure calls* [11,30]. The approach considers only crash-tolerant state machine replication schemes and requires that

both the client and server are aware of each other's configurations. No attempt is made at making the approach transparent, and the approach does not support secure or Byzantine-tolerant replication schemes. Other proposals for specialized protocols for interacting replicated service include [2,21,28]. Aegean [4] also considers the interaction between replicated services. It considers the blocking RPC model incompatible with the state machine replication model of Paxos and proposes a solution based on speculative execution. Compared to these approaches, Escher provides a secure and general-purpose approach to interacting services that can have undergone various forms of refinement, including any of a variety of replication protocols. Moreover, Escher supports dynamic updates of a service without having to reconfigure other services that it interacts with.

Automated Synthesis. Some prior work focuses on generating low-level implementations of distributed protocols from their high-level specifications. In [9], Bonakdarpour et al. give a framework for automatedly deriving the implementation of a distributed application from its component-based "Behavior, Interaction, Priority" model. DistAlgo [19] is a high-level specification language for distributed protocols that can be compiled into Python implementations. While Escher's wrappers can be used to deploy a distributed refinement of a specific agent that has an implementation, Escher is not concerned with automatically refining a specification. Instead, Escher focuses on providing uniform communication between refined components, which is key for supporting modular and dynamic systems.

5 Conclusion

As multi-tiered cloud applications have become commonplace, the time has come to have first-class support for interacting distributed services. This paper proposes Escher, a middleware that uses service refinement to ease the development and evolution of such systems of systems. Escher manages refinement hierarchies and provides a message bus that allows services to communicate irrespective of their refinement level. Escher also provides isolation between distributed services, so that each can be developed and evolved independently of others. For certain classes of services, Escher provides built-in refinements, such as replication for fault tolerance and sharding for scalability, that can be applied without having to write or change code.

Acknowledgments. We thank the reviewers for their feedback. This work was supported in part by a Google Faculty Research Award, and by the NSF grants CSR-17620155, CNS-CORE 2008667, and CNS-CORE 2106954.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2), 253–284 (1991)

2. Adya, A., et al.: Farsite: federated, available, and reliable storage for an incompletely trusted environment. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002). USENIX, December 2002
3. Ajmani, S., Liskov, B., Shrira, L.: Modular software upgrades for distributed systems. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 452–476. Springer, Heidelberg (2006). https://doi.org/10.1007/11785477_26
4. Aksoy, R.C., Kapritsos, M.: Aegean: replication beyond the client-server model. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, pp. 385–398, New York, NY, USA. Association for Computing Machinery (2019)
5. Altinbuken, D., Van Renesse, R.: Ovid: a software-defined distributed systems framework. In: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2016), June 2016
6. Ananthanarayanan, R., et al.: Photon: fault-tolerant and scalable joining of continuous data streams. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 577–588, New York, NY, USA. Association for Computing Machinery (2013)
7. Bernstein, D.: Containers and cloud: from LXC to Docker to Kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014)
8. Bidan, C., Issarny, V., Saridakis, T., Zarras, A.: A dynamic reconfiguration service for CORBA. In: Proceedings of the Fourth International Conference on Configurable Distributed Systems (Cat. No.98EX159) (1998)
9. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: A framework for automated distributed implementation of component-based models. Distrib. Comput. **25**, 10 (2012)
10. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Queue **14**(1), 70–93 (2016)
11. Cooper, E.C.: Replicated procedure call. SIGOPS Oper. Syst. Rev. **20**(1), 44–56 (1986)
12. IBM Cloud Education. ESB (Enterprise Service Bus). https://www.ibm.com/cloud/learn/esb
13. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)
14. Object Management Group. The Common Object Request Broker: Architecture and specification, revision 2.4.1, forma1/00-11-07, November 2000
15. Hawblitzel, C., et al.: IronFleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, pp. 1–17, New York, NY, USA. Association for Computing Machinery (2015)
16. Apache Kafka. https://kafka.apache.org/
17. Laibinis, L., Troubitsyna, E., Iliasov, A., Romanovsky, A.B.: Fault tolerant middleware for agent systems: a refinement approach. In: 12th European Workshop on Dependable Computing (EWDC 2009), Toulouse, France, May 2009
18. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
19. Liu, Y.A., Stoller, S.D., Lin, B.: High-level executable specifications of distributed algorithms. In: Richa, A.W., Scheideler, C. (eds.) SSS 2012. LNCS, vol. 7596, pp. 95–110. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33536-5_11

20. Horstmann, M., Kirtland, M.: DCOM architecture. In: Microsoft Developer Network, July 1997
21. Netto, H.V., Lung, L.C., Correia, M., Luiz, A.F., de Souza, L.M.S.: State machine replication in containers managed by Kubernetes. J. Syst. Archit. **73** (2017)
22. Envoy Proxy. https://www.envoyproxy.io/
23. TIBCO Enterprise Message Service. https://www.tibco.com/products/tibco-enterprise-message-service
24. Shoup, R.: Service architectures at scale: Lessons from Google and eBay
25. Truyen, E., Joosen, W., Verbaeten, P., Jorgensen, B.N.: On interaction refinement in middleware. In: Workshop on Component-Oriented Programming, June 2000
26. Valesky, T.: Enterprise JavaBeans: Developing Component-Based Distributed Applications. Addison-Wesley Longman Publishing Co., Inc., USA (1999)
27. Vieira, M.A.M., et al.: Fast packet processing with EBPF and XDP: concepts, code, challenges, and applications. ACM Comput. Surv. **53**(1) (2020)
28. Wang, Y., et al.: Robustness in the Salus scalable block store. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, NSDI 2013, pp. 357–370, USA. USENIX Association (2013)
29. Wilcox, J.R., et al.: Verdi: a framework for implementing and formally verifying distributed systems. SIGPLAN Not. **50**(6), 357–368 (2015)
30. Yap, K.S., Jalote, P., Tripathi, S.: Fault tolerant remote procedure call. In: Proceedings of the 8th International Conference on Distributed Computing Systems (1988)