Ovid: A Software-Defined Distributed Systems Framework to support Consistency and Change

Deniz Altınbüken, Robbert van Renesse
Department of Computer Science
Cornell University
{deniz,rvr}@cs.cornell.edu

Abstract

We present Ovid, a framework for building large-scale distributed systems that have to support strong consistency and at the same time need to be able to evolve quickly as a result of changes in their functionality or the assumptions they made for their initial deployment. In practice, organic growth often makes distributed systems increasingly more complex and unmanageable. To counter this, Ovid supports transformations*, automated refinements that allow distributed systems to be developed from simple components. Examples of transformations include replication, batching, sharding, and encryption. Refinement mappings prove that transformed systems implement the specification. The result is a* software-defined distributed system*, in which a logically centralized controller specifies the components, their interactions, and their transformations. Such systems can be updated on-the-fly, changing assumptions or providing new guarantees while keeping the original implementation of the application logic unchanged.*

1 Introduction

Building distributed systems is hard, and evolving them is even harder. A simple client-server architecture may not be able to scale to developing workloads. Sharding, replication, batching, and similar improvements will be necessary to incorporate over time. Next, it will be necessary to support online configuration changes as hardware is being updated or a new version of the system is being deployed. Online software updates will be necessary for bug fixes, new features, enhanced security, and so on. All this is sometimes termed "organic growth" of a distributed system. While we have a good understanding of how to build a strongly consistent service based on techniques such as state machine replication and atomic transactions, we do not have the technology to build *consistent systems* that are comprised of many services and that undergo organic growth.

In this paper we describe the design of *Ovid*, a framework for building, maintaining, and evolving distributed systems that have to support strong consistency. The framework leverages the concept of refinement [15] or, equivalently, backward simulation [20]. We start out with a relatively simple specification of *agents*. Each agent is a simple self-contained state machine that transitions in response to messages it receives and may produce output messages for other agents. Next, we apply *transformations* to agents such as replication or sharding. Each

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

transformation replaces an agent by one or more new agents. For each transformation we supply a refinement mapping [15] from the new agents to the original agent to demonstrate correctness, but also to be able to obtain the state of the original agent in case a reconfiguration is necessary. Transformations can be applied recursively, resulting in a tree of transformations.

A collection of agents itself is a state machine that transitions in response to messages it receives and may produce output messages for other agents. Consequently, a collection of agents can be considered an agent itself. When an agent is replaced by a set of agents, the question arises what happens to messages that are sent to the original agent. For this, each transformed agent has one or more *ingress agents* that receive such incoming messages. The routing is governed by routing tables: each agent has a routing table that specifies, for each destination address, what the destination agent is. Inspired by software-defined networks [7, 21], Ovid has a logically centralized controller, itself an agent, that determines the contents of the routing tables.

Besides routing, the controller determines where agents run. Agents may be co-located to reduce communication overhead, or run in different locations to benefit performance or failure independence. In order for the controller to make placement decisions, agents are tagged with location constraints. The result is what can be termed a "software-defined distributed system" in which a programmable controller manages a running system.

Ovid supports on-the-fly reconfiguration based on "wedging" agents [5, 1]. By wedging an agent, it can no longer make transitions, allowing its state to be captured and re-instantiated for a new agent. By updating routing tables the reconfiguration can be completed. This works even for agents that have been transformed, using the refinement mapping for retrieving state.

This paper is organized as follows. We start with discussing related work in Section 2. In Section 3 we present a system model for Ovid, including agents, communication, and transformation. Section 4 presents various examples of transformation. In Section 5 we discuss how Ovid supports online configuration changes. We describe how agents are scheduled and how routing is done in Section 6. We conclude in Section 7.

2 Related Work

There has been much work on automating distributed system creation and verification as well as implementing long-lived distributed systems that can function in evolving environments. In this section we look at both approaches and present related work.

2.1 Automated Distributed System Creation and Verification

Mace [13] is a language-based solution to automatically generate complicated distributed system deployments using high-level language constructs. Mace is designed as a software package that comprises a compiler that translates high-level service specifications to working C++ code. In Mace, a distributed system is represented as a group of nodes, where each node has a state that changes with message or timer events. To construct a distributed system using Mace, the user has to specify handlers, constants, message types, state variables and services in a high-level. The compiler then creates a working distributed application in C++ according to the specifications provided.

CrystalBall [28] is a system built on top of the Mace framework to verify a distributed system by exploring the space of executions in a distributed manner and having every node predict the outcome of their behavior. In CrystalBall, nodes run a state exploration algorithm on a recent consistent snapshot of their neighborhood and predict possible future violations of specified safety properties, in effect executing a model checker running concurrently with the distributed system. This is a more scalable approach compared to running a model checker from the initial state of a distributed system and doing exhaustive state exploration.

Similarly, other recent projects have been focusing on verifying distributed systems and their components automatically. In [23] Schiper et al. use the formal EventML [22] language to create specifications for a Paxos-

based broadcast protocol that can be formally verified in NuPRL [8]. This specification is then compiled into a provably correct and executable implementation automatically and used to build a highly available database.

In [26], Wilcox et al. present a framework, namely Verdi, for implementing practical fault-tolerant distributed systems and then formally verifying that the implementations meet their specifications. Verdi provides a Coq toolchain for writing executable distributed systems and verifying them, a mechanism to specify fault models as network semantics, and verified system transformers that take an existing system and transform it to another system that makes different assumptions about its environment. Verdi is able to transform systems to assume different failure models, even if it is not able to transform systems to provide new guarantees.

IronFleet [9] proposes building and verifying distributed systems using TLA-style state-machine refinements and Hoare-logic verification. IronFleet employs a language and program verification toolchain Dafny [16] that automates verification and it enables proving safety and liveness properties for a given distributed system.

Systems like CrystalBall, Verdi, and IronFleet and languages like EventML can be used in combination with Ovid to build provably correct large-scale infrastructure services that comprise multiple distributed systems. These systems can be employed to prove the safety and liveness properties of different modules in Ovid, as well as the distributed systems that are transformed by Ovid. This way, large-scale infrastructure systems that are built as a combination of multiple provably correct distributed systems can be constructed by Ovid.

2.2 Implementing Evolving Distributed Systems

One approach in implementing evolving distributed systems is building reconfigurable systems. Reconfigurable distributed systems [4, 6, 12, 14] support the replacement of their sub-systems. In [3], Ajmani et al. propose automatically upgrading the software of long-lived, highly-available distributed systems gradually, supporting multi-version systems. In the infrastructure presented, a distributed system is modeled as a collection of objects. An object is an instance of a class. During an upgrade old and new versions of a class and their instances are saved by a node and both versions can be used depending on the rules of the upgrade. This way, multi-versioning and modular upgrades are supported in the object-level. In their methodology, Ajmani et al. use transform functions that reorganizes a node's persistent state from the representation required by the old instance to that required by the new instance, but these functions are limited with transforming the state of a node, whereas we transform the distributed system as a whole.

Horus [25, 17] and Ensemble [10, 24] employ a modular approach to building distributed systems, using *micro-protocols* that can be combined together to create protocols that are used between components of a distributed system. Specific guarantees required by a distributed system can be implemented by creating different combinations of micro-protocols. Each micro-protocol layer handles some small aspect of guarantees implemented by a distributed system, such as fault-tolerance, encryption, filtering, and replication. Horus and Ensemble also support on-the-fly updates [18, 19].

Prior work has used refinement mappings to prove that a lower-level specification of a distributed system correctly implements a higher-level one. In [2], Aizikowitz et al. uses refinement mappings to show that a distributed, multiple-server implementation of a service is correct if it implements the high-level, single-server specification. Our work generalizes this idea to include other types of system transformations such as sharding, batching, replication, encryption, and so on.

3 System Model

3.1 Agents

A system consists of a set $\mathcal{A}$ of *agents* $\alpha, \beta, ...$ that communicate by exchanging messages. Each agent has a unique identifier. A message is a pair $\langle$*agent identifier*, *payload*$\rangle$. A message sent by a correct agent to another

```
agent KeyValueStore :
  var map
  initially : ∀k ∈ Key : map[k] = ⊥
  transition ⟨g, p⟩ filter g ≠ ⊥:
    if p.type = PUT:
      map[p.key] := p.value
    elif p.type = GET:
      SEND ⟨p.replyAgentID, map[p.key]⟩
```

Figure 1: Pseudocode for a key-value store agent. The key-value store keeps a mapping from keys to values and maps a new value to a given key with the PUT operation and returns the value mapped to a given key with the GET operation.

```
agent Client :
  var result
  transition ⟨g, p⟩:
    if g = ⊥:
      SEND⟨ 'KVS', ⟨type : GET,
                    key : 'foo',
                    replyAgentID : 'client'⟩⟩
    else:
      result := p
```

Figure 2: Pseudocode for a client agent that requests a key mapping from the key-value store agent with a GET operation on the key 'foo'.

correct agent is eventually delivered, but multiple messages are not necessarily delivered in the order sent and there is no bound on latency.

We describe the state of an agent by a tuple $(\mathcal{ID}, \mathcal{SV}, \mathcal{ST}, \mathcal{IM}, \mathcal{PM}, \mathcal{OM}, \mathcal{WB}, \mathcal{RT})$:

- $\mathcal{ID}$: a unique identifier for the agent;
- $\mathcal{SV}$: a collection of *state variables* and their values;
- $\mathcal{TF}$: a *transition function* invoked for each input message;
- $\mathcal{IM}$: a collection of input messages that have been received;
- $\mathcal{PM}$: a subset of $\mathcal{IM}$ of messages that have been processed;
- $\mathcal{OM}$: a collection of all output messages that have been produced;
- $\mathcal{WB}$: a *wedged bit* that, when set, prevents the agent from transitioning. In particular, no more input messages can be processed, no more output messages can be produced, and the state variables are immutable;
- $\mathcal{RT}$: a routing table that maps agent identifiers to agent identifiers.

Agents that are faulty make no more transitions (*i.e.*, their wedged bit is set) and in addition stop attempting to deliver output messages. Assuming its $\mathcal{WB}$ is clear, a correct agent eventually selects a message from $\mathcal{IM}\backslash\mathcal{PM}$ (if non-empty), updates the local state using $\mathcal{TF}$, and adds the message to $\mathcal{PM}$. In addition, the transition may produce one or more messages that are added to $\mathcal{OM}$. Optionally, the transition function may specify a *filter predicate* that specifies which of the input messages are currently of interest. Such transitions are atomic. $\mathcal{IM}$ initially consists of a single message $\langle \perp, \perp \rangle$ that can be used for the agent to send some initial messages in the absence of other input. Note that a message with a particular content can only be processed once by a particular agent; applications that need to be able to exchange the same content multiple times are responsible for adding additional information such as a sequence number to distinguish the copies.

See Figure 1 for an example of an agent that implements a key-value store: a mapping from keys to values. (Only $\mathcal{SV}$ and $\mathcal{TF}$ are shown.) For example, the transition is enabled if there is a PUT request in $\mathcal{IM}\backslash\mathcal{PM}$, and the transition simply updates the map but produces no output message. The agent identifier in the input message is ignored in this case. If there is a GET request in $\mathcal{IM}\backslash\mathcal{PM}$, the transition produces a response message to the agent identifier included in the payload p. The command **SEND** $\langle g', p' \rangle$ adds message $\langle g', p' \rangle$ to $\mathcal{OM}$. In both cases the request message is added to $\mathcal{PM}$.

```
agent ShardIngressProxy :
  transition ⟨g, p⟩:
    if p ≠ ⊥ ∧ P(p.key):
      SEND ⟨ 'KVS/ShardEgressProxy1', ⟨g, p⟩⟩
    elif p ≠ ⊥ ∧ ¬P(p.key):
      SEND ⟨ 'KVS/ShardEgressProxy2', ⟨g, p⟩⟩
```

Figure 3: Pseudocode for a client-side sharding ingress proxy agent for the key-value store. The ingress proxy agent mimics `'KVS'` to the client and forwards client requests to the correct shard depending on the key.

```
agent ShardEgressProxy :
  transition ⟨g', ⟨g, p⟩⟩:
    SEND ⟨g, p⟩
```

Figure 4: Pseudocode for a server-side sharding egress proxy agent for the key-value store. The egress proxy agent mimics the client to a shard of the key-value store and simply forwards a received client request.

Figure 2 gives an example of a client that invokes a `GET` operation on the key 'foo'. The routing table of the client agent must contain an entry that maps `'KVS'` to the identifier of the key-value store agent, and similarly, the routing table of the key-value store agent must contain an entry that maps `'client'` to the identifier of the client agent.

We note that our specifications are *executable*: every state variable is instantiated at an agent and every transition is local to an agent.

3.2 Transformation

The specification of an agent α can be *transformed*— replacing it with one or more new agents in such a way that the new agents collectively implement the same functionality as the original agent from the perspective of the other, unchanged, agents. In the context of a particular transformation, we call the original agent *virtual*, and the agents that result from the transformation *physical*.

For example, consider the key-value store agent of Figure 1. We can *shard* the virtual agent by creating two physical copies of it, one responsible for all keys that satisfy some predicate $P(key)$, and the other responsible for the other keys. That is, P is a binary hashing function. To glue everything together, we add additional physical agents: a collection of *ingress proxy agents*, one for each client, and two *egress proxy agents*, one for each server.[1] An ingress proxy agent mimics the virtual agent `'KVS'` to its client, while an egress proxy agent mimics the client to a shard of the key-value store. The routing table of the client agent is modified to route messages to `'KVS'` to the ingress proxy. Figures 3 and 4 present the code for these proxies. Note that the proxy agents have no state variables. Moreover, Figure 5 illustrates their configuration as a directed graph. Every physical agent is pictured as a separate node and agents that are co-located are shown in the same rectangle representing a box. The directed edges between nodes illustrate the message traffic patterns between agents. Lastly, the dotted line is used to separate two abstraction layers from each other.

A transformation is essentially a special case of a refinement in which an *executable* specification is refined to another executable specification. To show the correctness of refinement, one must exhibit:

- a mapping of the state of the physical agents to the state of the virtual agent,
- a mapping of the transitions of the physical agents to transitions in the virtual agent, or identify those transitions as *stutter transitions* that do not change the state of the virtual agent.

In this example, a possible mapping is as follows:

[1] For this particular example, it would be possible to not use server-side egress proxies and have the client-side ingress proxies send directly to the shards. The given solution is chosen to illustrate various concepts in our system.

- $\mathcal{ID}$: the identifier of the virtual key-value store agent is unchanged and constant;
- $\mathcal{SV}$: the map of the virtual agent is the union of the two physical shards;
- $\mathcal{TF}$: the transition function of the virtual agent is also as specified;
- $\mathcal{IM}$: the set of input messages of the virtual agent is the union of the sets of input messages of all client-side proxies;
- $\mathcal{PM}$: the set of processed messages of the virtual agent is the union of the set of processed messages of the two shards;
- $\mathcal{OM}$: the set of output messages of the virtual agent is the union of the sets of output messages of the two shards;
- $\mathcal{WB}$: the wedged bit is the logical 'and' of the wedged bits of the shard agents (when both shards are wedged, the original agent can no longer transition either);
- $\mathcal{RT}$: the routing table is a constant.

In addition, the transitions of physical agents map to transitions in the virtual agent as follows:

- receiving a message in one of the $\mathcal{IM}$s of the ingress proxy agents maps to a message being added to the $\mathcal{IM}$ of the virtual agent;
- each $\mathcal{TF}$ transition in the physical shards maps to the corresponding transition in the virtual key-value store agent. In addition, adding a message to either $\mathcal{PM}$ or $\mathcal{OM}$ in one of the shards maps to the same transition in the virtual agent;
- setting the $\mathcal{WB}$ of one of the shards so that both become set causes the $\mathcal{WB}$ of the virtual agent to become set;
- clearing the $\mathcal{WB}$ of one of the shards when both were set causes the $\mathcal{WB}$ of the virtual agent to become cleared;
- any other transition in the physical agents is a stutter.

The example illustrates *layering* and *encapsulation*, common concepts in distributed systems and networking. Figure 5 shows two layers with an abstraction boundary. The top layer shows an application and its clients. The bottom layer multiplexes and demultiplexes. This is similar to multiplexing in common network stacks. For example, the EtherType field in an Ethernet header, the protocol field in an IP header, and the destination port in a TCP header all specify what the next protocol is to handle the encapsulated payload. In our system, agent identifiers fulfill that role. Even if there are multiple layers of transformation, each layer would use, uniformly, an agent identifier for demultiplexing.

The example specifically illustrates sharding, but there are many other kinds of transformations that can be applied in a similar fashion, among which:

- *State Machine Replication*: similar to sharding, this deploys multiple copies of the original agent. The proxies in this case run a replication protocol that ensures that all copies receive the same messages in the same order;
- *Primary-Backup Replication*: this can be applied to applications that keep state on a separate disk using read and write operations. In our model, such a disk is considered a separate agent. Fault-tolerance can be achieved by deploying multiple disk agents, one of which is considered primary and the others backups;
- *Load Balancing*: also similar to sharding, and particularly useful for stateless agents, a load balancing agent is an ingress proxy agent that spreads incoming messages to a collection of server agents;

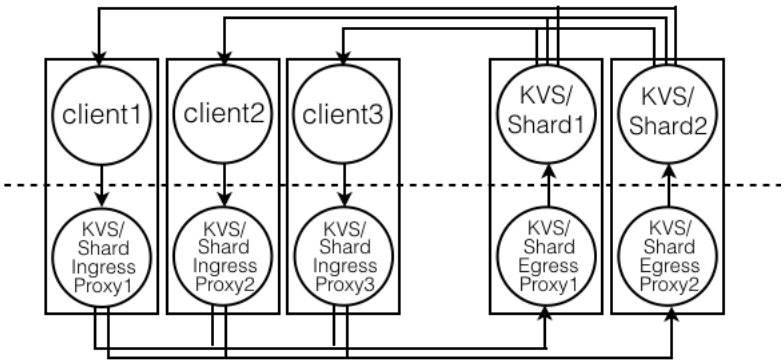


Figure 5: Configuration for the key-value store that has been transformed to be sharded two-ways.

- *Encryption, Compression, Batching, ...*: between any pair of agents, one can insert a pair of agents that encode and decode sequences of messages respectively;
- *Monitoring, Auditing*: between any pair of agents, an agent can be inserted that counts or logs the messages that flow through it.

Above we have presented transformations as refinements of individual agents. In limited form, transformations can sometimes also be applied to sets of agents. For example, a *pipeline of agents* (in which the output of one agent form the input to the next) acts essentially as a single agent, and transformations that apply to a single agent can also be applied to pipelines. Some transformations, such as Nysiad [11], apply to particular configurations of agents. For simplicity, we will focus here on transformations of individual agents only, but believe the techniques can be generalized more broadly.

3.3 Agent Identifiers and Transformation Trees

Every agent in the system has a unique identifier. The agents that result from transformation have identifiers that are based on the original agent's identifier, by adding new identifiers in a `'path name'` style. Thus an agent with identifier `'X/Y/Z'` is part of the implementation of agent `'X/Y'`, which itself is part of the implementation of agent `'X'`, which is a top level specification. In our running example, assume the identifier of the original key-value store is `'KVS'`. Then we can call its shards `'KVS/Shard1'` and `'KVS/Shard2'`. We can call the server proxies `'KVS/ShardEgressProxy1'` and `'KVS/ShardEgressProxy2'` respectively, and we can call the client proxies `'KVS/ShardIngressProxy1'`, `'KVS/ShardIngressProxy2'`, ...

The client agent in this example still sends messages to agent identifier `'KVS'`, but due to transformation the original `'KVS'` agent no longer exists physically. The client's routing table maps agent identifier `'KVS'` to `'KVS/ShardIngressProxy`*X*`'` for some *X*. Agent `'KVS/ShardIngressProxy`*X*`'` encapsulates the messages it received from its client and sends it to agent identifier `'KVS/ShardEgressProxy1'` or `'KVS/ShardEgressProxy2'` depending on the hash function. Assuming those proxy agents have not been transformed themselves, there is again a one-to-one mapping to from agent identifiers to corresponding agent identifiers. Each egress proxy ends up sending to agent identifier `'KVS'`. Agent identifier `'KVS'` is mapped to agent `'KVS/Shard1'` at agent `'KVS/ShardEgressProxy1'` and to agent `'KVS/Shard2'` at agent `'KVS/ShardEgressProxy2'`. Note that if identifier *X* in a routing table is mapped to an identifier *Y*, it is always the case that *X* is a prefix of *Y* (and is identical to *Y* in the case the agent has not been refined).

Given the original specification and the transformations that have been applied, it is always possible to determine the destination agent for a message sent by a particular source agent to a particular agent identifier.

```
agent KeyValueStore :
  var : map, counter
  initially : ∀k ∈ Key : map[k] = ⊥ ∧ counter = 0

  transition ⟨g, ⟨c, p⟩⟩ filter c = counter:
    if p.type = PUT:
      map[p.key] := p.value
    elif p.type = GET:
      SEND ⟨p.replyAgentID, map[p.key]⟩
    counter := counter + 1
```

Figure 6: Pseudocode for a determistic key-value store agent that handles requests in order using a counter.

```
agent Numberer :
  var counter
  initially : counter = 0

  transition ⟨g, p⟩ filter g ≠ ⊥:
    SEND ⟨g, ⟨counter, p⟩⟩
    counter := counter + 1
```

Figure 7: Pseudocode for the numbering agent that numbers every message before it forwards it to its destination.

This even works if agents are created dynamically. For example, if a new client `'client2'` is added to our example, and sends a message to agent identifier `'KVS'`, we can determine that agent `'KVS'` has been transformed and thus a new client-side ingress proxy agent has to be created, and appropriate agent identifier to agent identifier mappings must be added. The client's request can now be delivered to the appropriate shard through the ingress proxy agent. The shard sends the response to agent identifier `'client2'`. In this case the new client itself has not been transformed, and so the mapping for agent identifier `'client2'` at the shard can be set up to point directly to agent `'client2'`. Should agent `'client2'` itself have been transformed, then upon the KVS shard sending to agent identifier `'client2'` the appropriate proxy agents can be instantiated on the server-side dynamically as well.

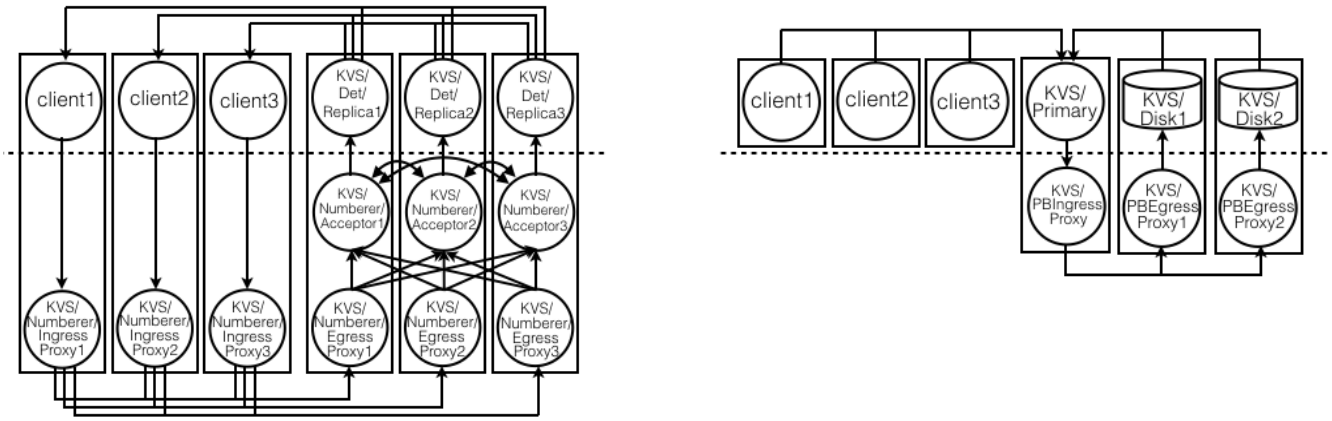
We represent the specification of a system and its transformation in a *Logical Agent Transformation Tree* (LATT). A LATT is a directed tree of agents. The root of this tree is a virtual agent that we call the *System Agent*. The "children" of this root are the agents before transformation. Each transformation of an agent (or set of agents) then results in a collection of children for the corresponding node.

This technique presents, to the best of our knowledge, the first general solution to composing transformed agents, such as a replicated client interacting with a replicated server. While certain special case solutions exist (for example, the Replicated Remote Procedure Call in the Circus system [27]), it has not been clear how, say, a client replicated using the replicated state machine approach would interact with a server that is sharded, with each shard chain replicated. At the same time, another client may have been transformed in another fashion, and also has to be able to communicate with the same server. The resulting mesh of proxies for the various transformations is complex and difficult to implement correctly "by hand." The Ovid framework makes composition of transformed agents relatively easy.

3.4 Ordering

In our system, messaging is reliable but not ordered. The reason is clear from the running example: even if input and output were ordered, that ordering is lost (and unnecessary) if the key-value store is sharded. Maintaining the output ordering would require additional complexity. We did not want to be tied to maintaining a property that would be hard to maintain end-to-end in the face of transformations.

However, for certain transformations it is convenient if, or even necessary that, input and output are ordered. A canonical example is state machine replication, which only works if each replica processes its input in the same order. Agents that need such ordering should require, for example, that messages are numbered or tagged with some kind of ordering dependencies. In case there cannot be two different messages with the same number,



(a) KVS transformed with state machine replication.

(b) KVS transformed with primary-backup replication.

Figure 8: The key-value store can be transformed to tolerate failures using different replication techniques.

an agent will make deterministic transitions. For example, Figure 6 shows the code for a deterministic version of the key-value store that can be replicated using state machine replication. If unreplicated, messages can be numbered by placing a *numbering agent* (Figure 7) in front of it that numbers messages from clients. When replicated with a protocol such a Paxos, Paxos essentially acts as a fault-tolerant numbering agent.

We can consider the pair of agents that we have created a refinement of the original `'KVS'` agent, and identify them as `'KVS/Deterministic'` and `'KVS/Numberer'`. By having clients map `'KVS'` to `'KVS/Numberer'`, their messages to `'KVS'` are automatically routed to the numbering agent.

4 Transformation Examples

Agents can be transformed in various ways and combined with other agents in various ways, resulting in complex but functional systems. In this section we go through some examples of how agents can be transformed to obtain fault-tolerance, scalability, and security.

To visualize transformations and the resulting configuration of a system, we use graphs with directed edges. The edges represent the message traffic patterns for a particular client and a particular server. Messages emanating from other clients, which themselves may be transformed in other ways, may not follow the same paths. The dotted lines are used to separate different abstraction layers from each other.

4.0.1 State Machine Replication

State Machine Replication is commonly used to change a non-fault tolerant system to a fault-tolerant one by creating replicas that maintain the application state. To ensure consistency between the replicas, they are updated using the same ordered inputs. As alluded to before, we support state machine replication by two separate transformations. First, we transform a deterministic agent simply by generating multiple copies of it. Second, we refine the numbering agent and replace it with a state machine replication protocol such as Paxos.

We start with the deterministic key-value store agent in Figure 6 and create copies of it. We can call its replicas `'KVS/Deterministic/Replica1'`, `'KVS/Deterministic/Replica2'`, and so on, but note that they each run identical code to the virtual `'KVS/Deterministic'` agent. Next, we take the `'KVS/Numberer'` agent, and replace it with a fault-tolerant version. For example, in order to tolerate f acceptor failures using the Paxos protocol, we may deploy `'KVS/Numberer/AcceptorX'` for $X \in [0, 2f]$.

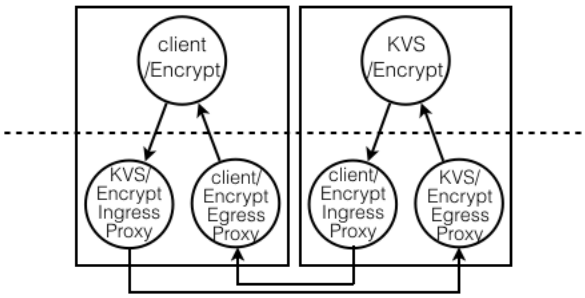


Figure 9: The key-value store can be transformed to accept encrypted traffic from clients by adding an ingress proxy on the client-side that encrypts client messages before they are sent and an egress proxy on the server-side that decrypt a message using the key shared between proxies. The reverse traffic is encrypted by transforming the clients in the same fashion.

As before, we will also deploy a client-side ingress proxy `'KVS/Numberer/IngressProxy`*C*`'` for each client *C*. Figure 8a shows the resulting system.

The technique can be combined with sharding. For example, we can first shard the key-value store and then replicate each of the shards individually (or a subset of the shards if we so desired for some reason). Alternatively, we can replicate the key-value store first, and then shard the replicas. While the latter is less commonly applied in practice, it makes sense in a setting where there are multiple heterogeneous data centers. One can place a replica in each datacenter, but use a different number of shards in each datacenter. And if one so desired, one could shard the shards, replicate the replicas, and so on.

4.0.2 Primary-Backup Replication

In primary-backup replication, a primary front-end handles requests from clients and saves the application state on multiple backup disks before replying back to the clients. When the primary fails, another primary is started with the state saved in one of the backup disks and continues handling client requests. To transform the key-value store to a primary-backup system, our example needs a new level of indirection, where the front-end KVS agent that handles the client requests itself is the client of a back-end disk that stores the application state. This new layering is necessary to introduce multiple back-up disks.

Accordingly, the primary-backup transformation of the KVS agent creates new proxies that enable the primary KVS agent, denoted as `'KVS/Primary'` to refer to backup disks as `'disk'` by having an entry in its routing table that maps `'disk'` to `'KVS/PBIngressProxy'`. Figure 8b shows this transformation. The `'KVS/PBIngressProxy'` can then send the update to be stored on disk to the disk proxies, denoted `'KVS/PBEgressProxy`*X*`'`, which in turn store the application state on their local back-end disk `'KVS/Disk`*X*`'`. This way, the key-value store is transformed with primary-backup replication without requiring the clients and the KVStore agents to change.

Similar to state machine replication, the fault-tolerance level of the transformed KVStore agent depends directly on the guarantees provided by primary-backup replication and the number of back-end disks that are created. As a result, because primary-backup replication can tolerate *f* failures with *f+1* back-end disks, the transformed KVStore in Figure 8b can tolerate the failure of one of the back-end disks.

4.0.3 Encryption, Compression, and Batching

One type of transformation that is supported by Ovid is adding an encoder and decoder between any two agents in a system, in effect processing streams of messages between these agents in a desired way. This transformation can be used to transform any existing system to support encryption, batching, and compression.

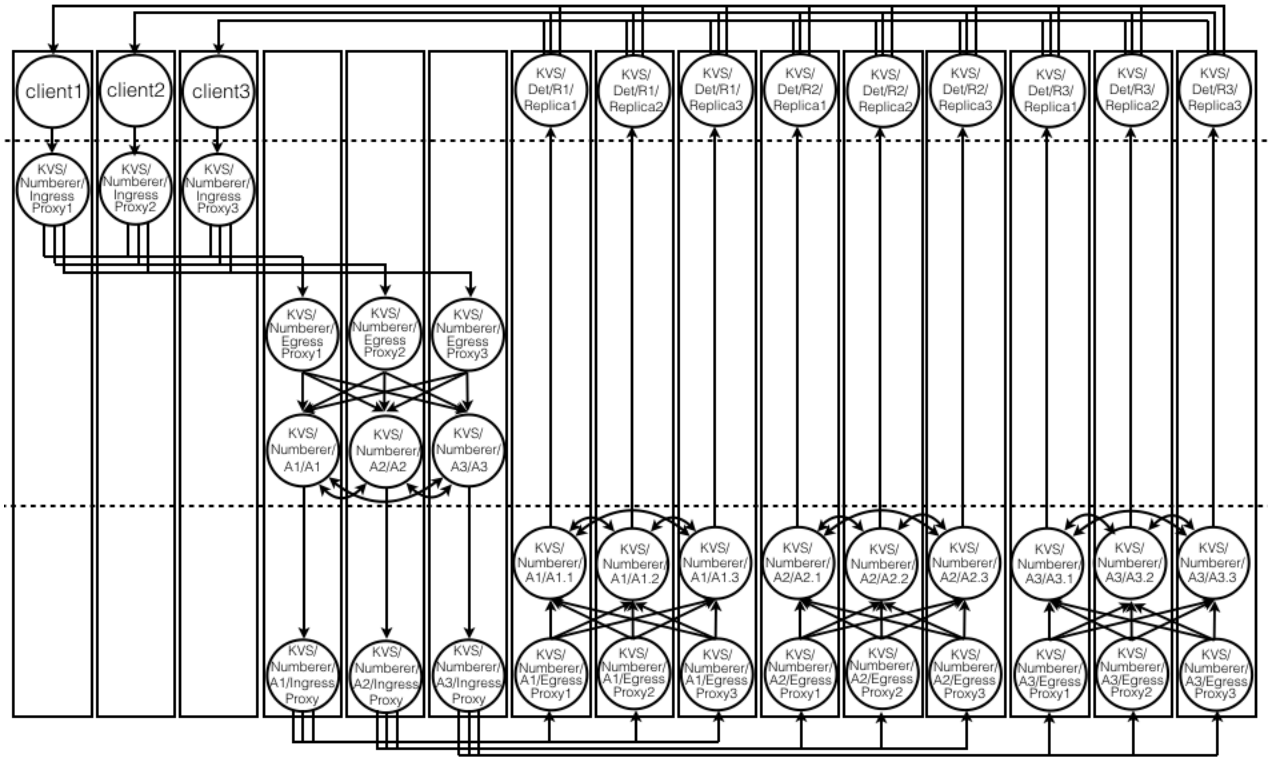


Figure 10: A crash fault-tolerant key-value store can be made to tolerate Byzantine failures by applying the Nysiad transformation, which replaces the replicas of the key-value store agent.

The encryption transformation is an example of these types of transformations. Encryption can be used to implement secure distributed systems by making traffic between different components unreadable to unauthorized components. To implement encryption in a distributed system, the requests coming from different clients can be encrypted and decrypted using unique encryption keys for clients. The transformation for encryption in Ovid follows this model and creates secure channels between different agents by forwarding messages to encryption and decryption proxies that are created during the transformation.

Figure 9 shows how the key-value store agent is transformed to support secure channels between the key-value store and the clients. Note that, in this example the traffic from the key-value store to client is encrypted, as well as the traffic from the client to the key-value store. When the client sends a message to the key-value store, the message is routed to `'KVS/EncryptIngressProxy'`, where it is encrypted and sent to `'KVS/EncryptEgressProxy'`. The egress proxy decrypts the message using the key shared between the proxies and forwards it to the key-value store to be handled. Virtually, `'KVS/EncryptIngressProxy'` and `'KVS/EncryptEgressProxy'` are separate entities than the client and the key-value store, but physically `'KVS/EncryptIngressProxy'` is co-located with the client, and `'KVS/EncryptEgressProxy'` is co-located with the key-value store shown as `'KVS/Encrypt'`. After the request is handled by the key-value store and a reply is sent back to the client, the reply follows a route symmetrical to the one from the client to the key-value store, since the client is transformed in the same fashion.

Batching and compression follow the same method: To achieve better performance in the face of changing load, multiple requests from a client can be batched together or compressed by an encoding agent and sent through the network as a single request. Then on the server side, these requests can be unbatched or decompressed accordingly and handled.

4.0.4 Crash Fault-Tolerance to Byzantine Fault-Tolerance with Nysiad

Many evolving distributed systems need to be transformed multiple times to change the assumptions they make about their environment or to change the guarantees offered by the system. For instance, a key-value store that has been transformed to handle only crash failures would not be able to handle bit errors in a large datacenter deployment. Ovid can solve this problem by transforming the crash fault-tolerant key-value store to tolerate Byzantine failures using the Nysiad [11] transformation. Figure 10 shows the transformation of the crash fault-tolerant key-value store of Figure 8a to a Byzantine fault-tolerant key-value store. This transformation replaces the replicas of the deterministic key-value store agent, namely `'KVS/Deterministic/Replica1'`, `'KVS/Deterministic/Replica2'`, and `'KVS/Deterministic/Replica3'`. A Byzantine failure is now masked as if it were a crash failure of a deterministic key-value store agent replica.

5 Evolution

A deployed system evolves. There can be many sources of such evolution, including:

- client agents come and go;
- in the face of changing workloads, applications may be elastic, and server agents may come an go as well;
- new functionality may be deployed in the form of new server agents;
- existing agents may be enhanced with additional functionality;
- bug fixes also lead to new versions of agents;
- the performance or fault tolerance of agents may be improved by applying transformations to them;
- a previously transformed agent may be "untransformed" to save cost;
- new transformations may be developed, possibly replacing old ones.

Our system supports such evolution, even as it happens "on-the-fly." As evolution happens, various correctness guarantees must be maintained. Instrumental in this are agents' wedged bits: we have the ability to temporarily halt an agent, even a virtual one. While halted, we can extract its state, transfer it to a new agent, update routing tables, and restart by unwedging the new agent. It is convenient to think of *versions* of an agent as different refinements of a higher level specification. Similarly, an old version has to be wedged, and state has to be transferred from an old version of an agent to a new one. The identifiers of such new agents should be different from old ones. We do this by incorporating version numbers into the identifiers of agents. For example, `'KVS:v2/IngressProxy:v3:2'` would name the second incarnation of version 3 of an ingress proxy agent, and this refines version 2 of the virtual KVS agent.

Obtaining the state of a wedged physical agent is trivial, and starting a physical agent with a particular state is also a simple operation. However, a wedged virtual agent may have its state distributed among various physical agents and thus obtaining or initializing the state is not straightforward. And even if a virtual agent is wedged, not all of its physical agents are necessarily wedged. For example, in the case of the sharded key-value store, the virtual agent is wedged if all shards are wedged, but it does not require that the proxies are wedged. Note also that there may be multiple levels of transformation: in the context of one transformation an agent may be physical, but in the context of another it may be virtual. For example, the shards in the last example may be replicated.

We require that each transformation provide a function *getState* that obtains the state of a virtual agent given the states of its physical agents. It is not necessarily the case that all physical agents are needed. For example, in

the case of a replicated agent, some physical agents may be unavailable and it should not be necessary to obtain the state of the virtual agent. Given the state of a virtual agent, a transformation also needs to provide a function that instantiates the physical agents and initializes their state.

While this solution works, it can lead to a significant performance hiccup. We are designing a new approach where the new physical agents can be started without an initial state before the virtual agent is even wedged. Then the virtual agent is wedged (by wedging the physical agents of its original transformation). This is a low-cost operation. Upon completion, the new physical agents are unwedged and "demand-load" state as needed from the old physical agents. Once all state has been transferred, the old physical agents can be shut down and garbage collected. This is most easily accomplished if the virtual agent has state that is easily divided into smaller, functional pieces. For example, in the case of a key-value store, each key-value pair can be separately transferred when needed.

6 Running Agents

6.1 Boxing

An agent is a logical location, but it is not necessary to run each agent on a separate host. Figure 5 illustrates *boxing*: co-locating agents in some way, such as on the same machine or even within a single process.

A *box* is an execution environment or virtual machine for agents. For each agent, the box keeps track of the agent's attributes and runs transitions for messages that have arrived. In addition, each box runs a *Box Manager Agent* that supports management operations such as starting a new agent or updating an agent's routing table.

Boxes also implement the reliable transport of messages between agents. A box is responsible for making sure that the set of output messages of an agent running on the box is transferred to the sets of input messages of the destination agents. For each destination agent, this assumes there is an entry for the message's agent identifier in the source agent's routing table, and the box also needs to know how to map the destination agent identifier to one or more network addresses of the box that runs the destinating agent. A box maintains the Box Routing Table (BRT) for this purpose. For now, we assume that each BRT is filled with the necessary information.

The box transmits the message to the destination box. Upon arrival, the destination box determines if the destination agent exists. If so, the box adds the message to the end of the agent's input messages, and returns an acknowledgment to the source box. The source box keeps retransmitting (without timeout) the message until it has received an acknowledgment. This process must restart if there is a reconfiguration of the destination agent. An output message can be discarded only if it is known that the message has been processed by the destination agent—being received is not a safe condition.

6.2 Placement

While resulting in more efficient usage of resources, boxing usually leads to a form *fate sharing*: a problem with a box such as a crash is often experienced by all agents running inside the box. Fate sharing makes sense for agents and its proxies, but it would be unwise to run replicas of the same agent within the same box.

We use *agent placement annotations* to indicate placement desirables and constraints. For example, agent α can have a set of annotations of the following form:

- α.SameBoxPreferred(β): β ideally is placed on the same box as α;
- α.SameBoxImpossible(β): β cannot run on the same box as α.

Placing agents in boxes can be automatically performed based on such annotations using a constraint optimizer. Placement is final: we do not currently consider support for *live migration* of agents between boxes. It is,

however, possible to start a new agent in another box, migrate the state from an old agent, and update the routing tables of other agents.

6.3 Controller Agent

So far we have glossed over several administrative tasks, including:

- What boxes are there, and what are their network addresses?
- What agents are there, and in which boxes are they running?
- How do agents get started in the first place?
- How do boxes know which agents run where?

As in other software-defined architectures, we deploy a logically centralized controller for administration. The controller itself is just another agent, and has identifier "controller". The controller agent itself can be refined by replication, sharding and so on. For scale, the agent may also be hierarchically structured. But for now we will focus on the high-level specification of a controller agent before transformation. In other words, for simplicity we assume that the controller agent is physically centralized and runs on a specific box.

As a starting point, the controller is configured with the LATT (Logical Agent Transformation Tree), as well as the identifiers of the box managers on those boxes. The BRT (Box Routing Table) of the box in which the controller runs is configured with the network addresses of the box managers.

First, the controller sends a message to each box manager imparting the network addresses of the box in which the controller agent runs. Upon receipt, the box manager adds a mapping for the controller agent to its BRT. (If the controller had been transformed, the controller would need to send additional information to each box manager and possibly instantiate proxy agents so the box manager can communicate with the controller.)

Using the agent placement annotations, the controller can now instantiate the agents of the LATT. This may fail if there are not enough boxes to satisfy the constraints specified by the annotations. Instantiation is accomplished by the controller sending requests to the various box managers.

Initially, the agents' routing tables start out containing only the "controller" mapping. When an agent sends a message to a particular agent identifier, there is an "agent identifier miss" event. On such an event, the agent ends up implicitly sending a request to the controller asking it to resolve the agent identifier to agent identifier binding. The controller uses the LATT to determine this binding and responds to the client with the destination agent identifier. The client then (again, implicitly) adds the mapping to its routing table.

Next, the box tries to deliver the message to the destination agent. To do this, the box looks up the destination agent identifier in its BRT, and may experience a "BRT miss". In this case, the box sends a request to the controller agent asking to resolve that binding as well. The destination agent may be within the same box as the source agent, but this can only be learned from the controller. One may think of routing tables as caches for the routes that the controller decides.

7 Conclusion

Ovid is an agent-based software-defined distributed systems platform that supports the complexity involved in building and maintaining large-scale distributed systems that have to support consistency and organic evolution. Starting from a simple specification, transformations such as replication and sharding can be applied to satisfy performance and reliability objectives. Refinement mappings allow reasoning about correctness, and also support capturing state from distributed components that have to be updated or replaced. A logically centralized controller simplifies the management of the entire system and allows sophisticated placement policies to be implemented.

We currently have a prototype implementation of Ovid that supports agents and transformations written in Python. Next, we want to develop tools to verify performance and reliability objectives of a deployment. We then plan to do evaluations of various large-scale distributed systems developed using Ovid.

Acknowledgments

The authors are supported in part by AFOSR grants FA2386-12-1-3008, F9550-06-0019, by the AFOSR MURI Science of Cyber Security: Modeling, Composition, and Measurements as AFOSR grant FA9550-11-1-0137, by NSF grants CNS-1601879, 0430161, 0964409, 1040689, 1047540, 1518779, 1561209, and CCF-0424422 (TRUST), by ONR grants N00014-01-1-0968 and N00014-09-1-0652, by DARPA grants FA8750-10-2-0238 and FA8750-11-2-0256, by MDCN/iAd grant 54083, and by grants from Microsoft Corporation, Infosys, Google, Facebook Inc., and Amazon.com.

References

[1] Hussam Abu-Libdeh, Robbert van Renesse, and Ymir Vigfusson. Leveraging sharding in the design of scalable replication protocols. In *Proceedings of the Symposium on Cloud Computing*, SoCC '13, Farmington, PA, USA, October 2013.

[2] Jacob I. Aizikowitz. *Designing Distributed Services Using Refinement Mappings*. PhD thesis, Cornell University, Ithaca, NY, USA, August 1989. Also available as technical report TR 89-1040.

[3] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 452–476, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] Christophe Bidan, Valrie Issarny, Titos Saridakis, and Apostolos Zarras. A dynamic reconfiguration service for corba. In *4th International Conference on Distributed Computing Systems*, ICCDS'98, pages 35–42. IEEE Computer Society Press, 1998.

[5] Ken Birman, Dahlia Malkhi, and Robbert van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.

[6] Toby Bloom. Dynamic module replacement in a distributed programming system. Technical Report MIT-LCSTR-303, MIT, 1983.

[7] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 1–12, New York, NY, USA, 2007. ACM.

[8] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[9] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*, SOSP '15. ACM, October 2015.

[10] Mark Garland Hayden. *The Ensemble System*. PhD thesis, Cornell University, Ithaca, NY, USA, 1998. AAI9818467.

[11] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, pages 175–188, Berkeley, CA, USA, 2008. USENIX Association.

[12] Christine R. Hofmeister and James M. Purtilo. A framework for dynamic reconfiguration of distributed programs. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, ICDCS 1991, pages 560–571, 1991.

[13] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[14] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[15] Leslie Lamport. Specifying concurrent program modules. *Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[16] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, Germany, 2010. Springer-Verlag.

[17] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Ken Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating System Principles*, Kiawah Island Resort, SC, USA, December 1999.

[18] Xiaoming Liu and Robbert van Renesse. Fast protocol transition in a distributed environment. In *19th ACM Conference on Principles of Distributed Computing (PODC 2000)*, Portland, OR, USA, July 2000.

[19] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta properties. In *International Workshop on Applied Reliable Group Communication at the International Conference on Distributed Computing Systems*, ICDCS 2011, Phoenix, AZ, USA, April 2001.

[20] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: Ii. timing-based systems. *Information and Computation*, 128(1):1–25, 1996.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, March 2008.

[22] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using eventml. *ECEASST*, 72, 2015.

[23] Nicolas Schiper, Vincent Rahli, Robbert van Renesse, Marck Bickford, and Robert L. Constable. Developing correctly replicated databases using formal tools. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 395–406, Washington, DC, USA, 2014. IEEE Computer Society.

[24] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. *Software Practice and Experience*, 28(9):963–979, August 1998.

[25] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[26] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed system. In *Proceedings of the 36th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI 2015, Portland, OR, USA, June 2015.

[27] J.C.P. Woodcock and Carroll Morgan. Refinement of state-based concurrent systems. In *VDM '90 VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 340–351. Springer Berlin Heidelberg, 1990.

[28] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 229–244, Berkeley, CA, USA, 2009. USENIX Association.