Ovid
A Software-Defined Distributed Systems Framework

Deniz Altinbuken, Robbert van Renesse
Cornell University
Ovid

Build distributed systems that are

- easy to evolve
- easy to reason about
- easy to compose
Approach

Create a theoretical model using abstractions

Deploy the distributed system using this model
Approach

Create a theoretical model using abstractions

Deploy the distributed system using this model
not fault tolerant
transform data store agent to be fault tolerant

\[ \text{replicate(datastore,3)} = \]
transform data store agent to be fault tolerant

replicate(datastore, 3) = datastore*3
A minority of replicas can fail in a crash fault tolerant asynchronous environment linearizable with the Paxos consensus protocol.

\[
\text{replicate(datastore,3)} = R^*3 + \text{datastore}^*3
\]
minority of replicas can fail

crash fault tolerant

asynchronous environment

linearizable

replicate(datastore,3) = proxy + R*3 + datastore*3
How do we know that a transformation did not break an agent?
How do we know that a transformation did not break an agent?
Use refinement mappings to prove equivalence.
client

data store
Use path names to represent transformations
Supported transformations

- replication
- byzantine resistance
- batching
- sharding
- encryption/decryption
- compression
- load-balancing
- deduplication
Approach

Create a theoretical model using abstractions

Deploy the distributed system using this model
Deployment

• Create a configuration from the model
• Deploy processes on boxes depending on the configuration
Deployment

- Create a configuration from the model
- Deploy processes on boxes depending on the configuration
Deployment

Machine Configuration

<table>
<thead>
<tr>
<th>Component</th>
<th>IP Address</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td>192.168.7.56</td>
<td>client.cpp</td>
</tr>
<tr>
<td>proxy</td>
<td>192.168.7.56</td>
<td>proxy.cpp</td>
</tr>
<tr>
<td>R1</td>
<td>192.168.7.80</td>
<td>replica.cpp</td>
</tr>
<tr>
<td>R2</td>
<td>192.168.7.81</td>
<td>replica.cpp</td>
</tr>
<tr>
<td>R3</td>
<td>192.168.7.82</td>
<td>replica.cpp</td>
</tr>
<tr>
<td>data store1</td>
<td>192.168.7.80</td>
<td>datastore.cpp</td>
</tr>
<tr>
<td>data store2</td>
<td>192.168.7.81</td>
<td>datastore.cpp</td>
</tr>
<tr>
<td>data store3</td>
<td>192.168.7.82</td>
<td>datastore.cpp</td>
</tr>
</tbody>
</table>
Deployment

Machine Configuration

- **client**: 192.168.7.56 client.cpp
- **proxy**: 192.168.7.56 proxy.cpp
- **R1**: 192.168.7.80 replica.cpp
- **R2**: 192.168.7.81 replica.cpp
- **R3**: 192.168.7.82 replica.cpp
- **data store1**: 192.168.7.80 datastore.cpp
- **data store2**: 192.168.7.81 datastore.cpp
- **data store3**: 192.168.7.82 datastore.cpp
Dynamic Routing

- Create routing tables
- Route messages to destination depending on the model
Dynamic Routing

Routing Configuration
client datastore proxy
proxy datastore R1,R2,R3
R1 datastore datastore1
R2 datastore datastore2
R3 datastore datastore3
datastore1 client client
datastore2 client client
datastore3 client client
Dynamic Routing

msg: {dest|payload}

controller

client → proxy → R1 → data store1 → R2 → data store2 → R3 → data store3
Dynamic Routing

msg: dest payload

data store 1
R1
R2
R3
data store 2
data store 3

data store
put(1, 6)

controller

client

proxy
Dynamic Routing

- Client
  - Data store: put(1, 6)

- Proxy

- Controller

- $R_1$ -> Data store$_1$
- $R_2$ -> Data store$_2$
- $R_3$ -> Data store$_3$

- msg: dest | payload

- send msg(dest, payload)
Dynamic Routing

msg: [dest|payload]

- send msg(dest,payload)
- look for dest in routing table
Dynamic Routing

- send msg(dest, payload)
- look for dest in routing table
- dest not present
Dynamic Routing

• send msg(dest,payload)
• look for dest in routing table
• dest not present
• send lookup message to controller
Dynamic Routing

msg: \[\text{dest} \mid \text{payload}\]

- send msg(dest,payload)
- look for dest in routing table
- dest not present
- send lookup message to controller
**Dynamic Routing**

- send msg(dest,payload)
- look for dest in routing table
- dest not present
- send lookup message to controller

**Routing Configuration**

- **client** datastore proxy
- **proxy** datastore R1,R2,R3
- **R1** datastore datastore1
- **R2** datastore datastore2
- **R3** datastore datastore3
- **datastore1** client client
- **datastore2** client client
- **datastore3** client client
Dynamic Routing

- send msg(dest,payload)
- look for dest in routing table
- dest not present
- send lookup message to controller
- get route mapping
Dynamic Routing

- send msg(dest,payload)
- look for dest in routing table
- dest not present
- send lookup message to controller
- get route mapping

msg:  

```
data store 1
R1
client
client
data store
put(1, 6)
controller
proxy
R2
data store 2
R3
data store 3
```
Dynamic Routing

- Client sends message (dest, payload)
- Look for dest in routing table
- If dest not present, send lookup message to controller
- Get route mapping
- Update routing table

```
data store
put(1, 6)

controller

Client → Proxy → [R1, R2, R3] → Data Stores
```
Dynamic Routing

- send msg(dest,payload)
- look for dest in routing table
- dest not present
- send lookup message to controller
- get route mapping
- update routing table
- send msg(dest,payload)
Dynamic Routing

msg: dest payload

- send msg(dest, payload)
- look for dest in routing table
- dest not present
- send lookup message to controller
- get route mapping
- update routing table
- send msg(dest, payload)
Dynamic Routing

- send msg(dest,payload)
- look for dest in routing table
- dest not present
- send lookup message to controller
- get route mapping
- update routing table
- send msg(dest,payload)
Conclusion

- Ovid introduces new abstractions and a new way of modeling distributed systems.

- Ovid can create distributed systems that can be reconfigured and deployed on the fly.

- Ovid makes building, running, maintaining and evolving distributed systems an easy task.