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

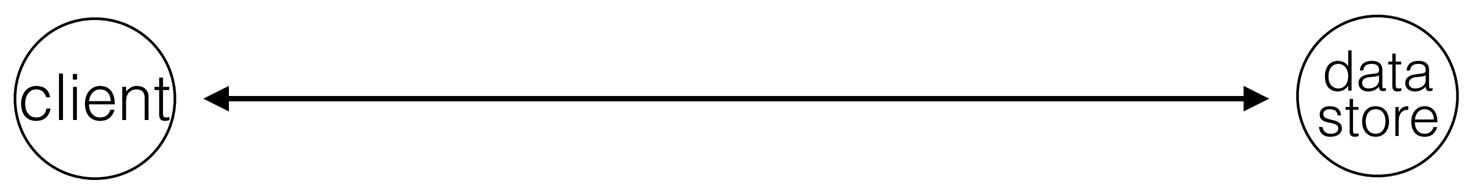
Create a theoretical model using abstractions

Deploy the distributed system using this model

Approach

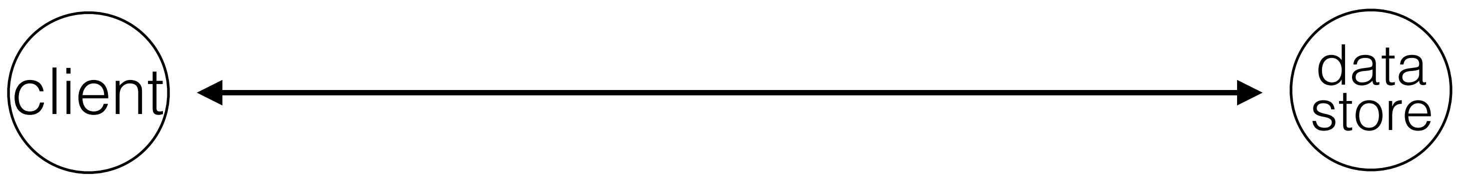
Create a theoretical model using abstractions

Deploy the distributed system using this model

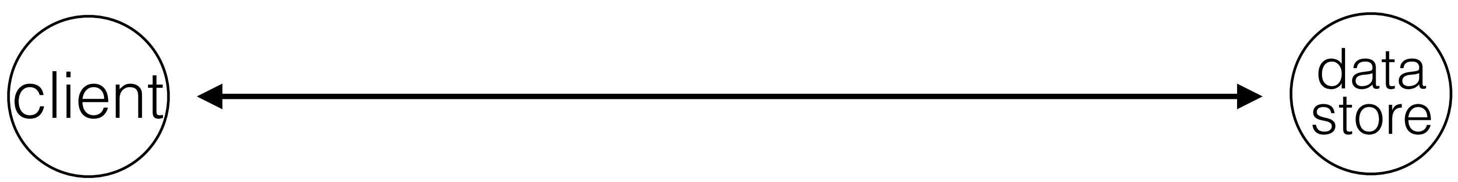

client
data
store


client
agent
data
store
agent

not fault tolerant

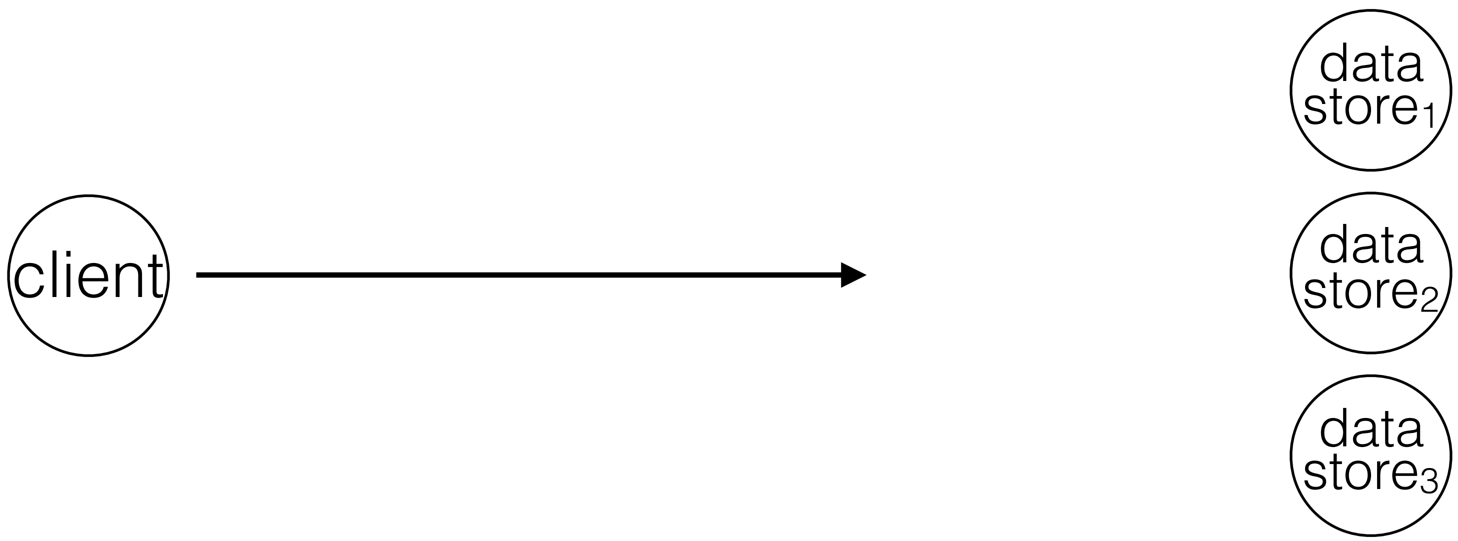


transform data store agent to be fault tolerant



replicate(datastore,3) =

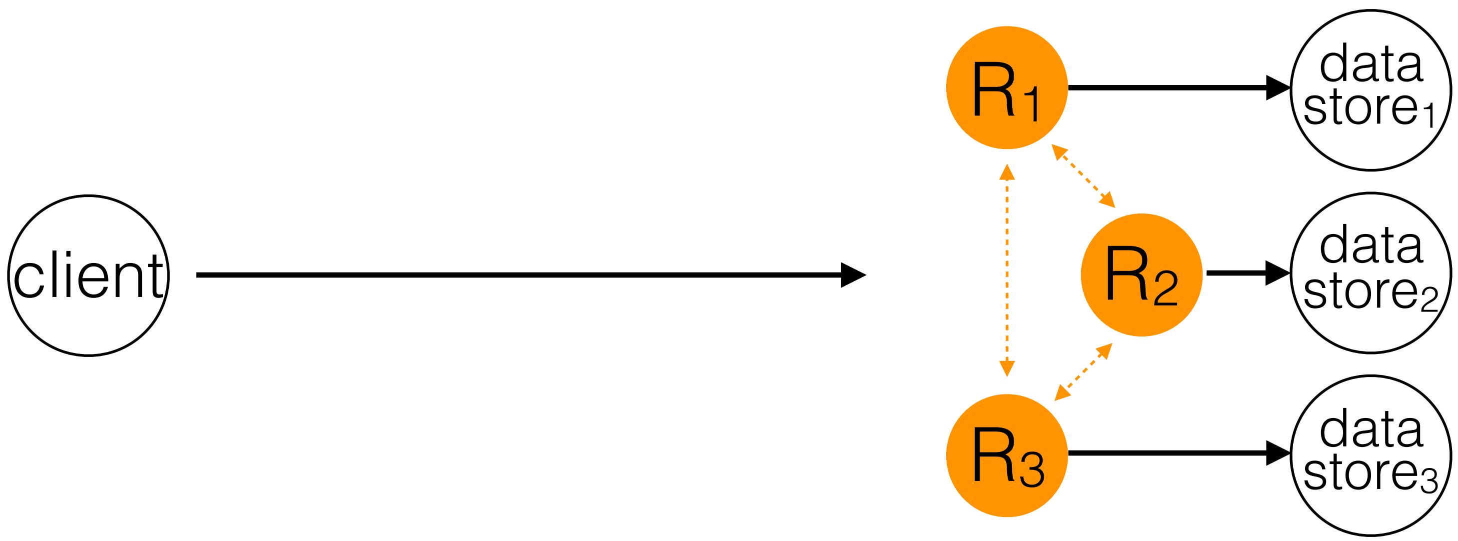
transform data store agent to be fault tolerant



replicate(datastore,3) = datastore*3

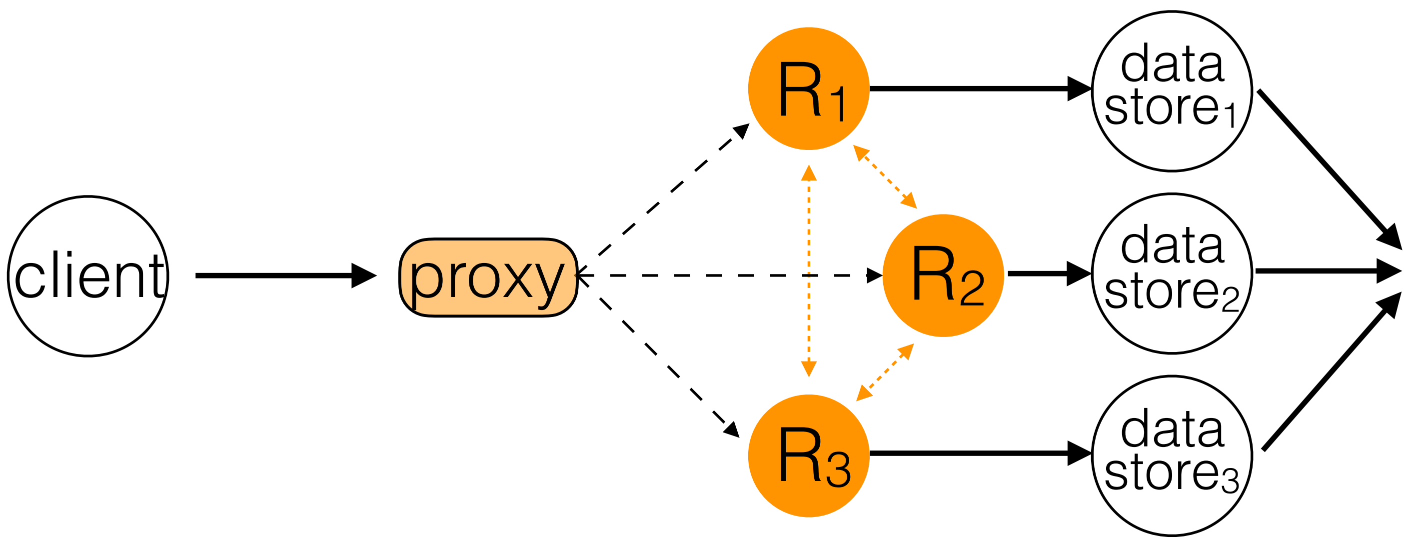
minority of replicas can fail
crash fault tolerant
asynchronous environment
linearizable

} Paxos consensus protocol



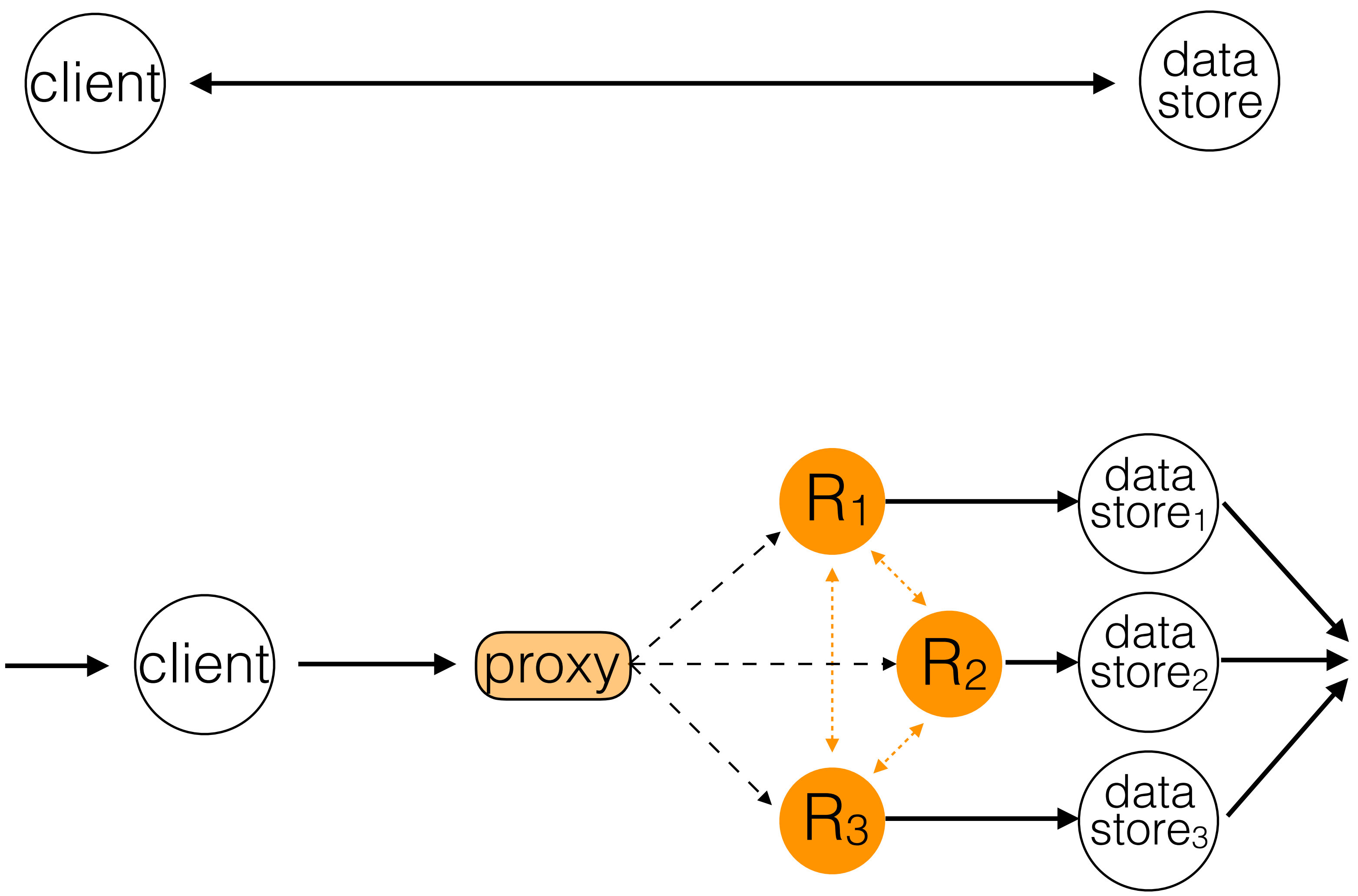
replicate(datastore,3) = R*3 + datastore*3

minority of replicas can fail
crash fault tolerant
asynchronous environment
linearizable

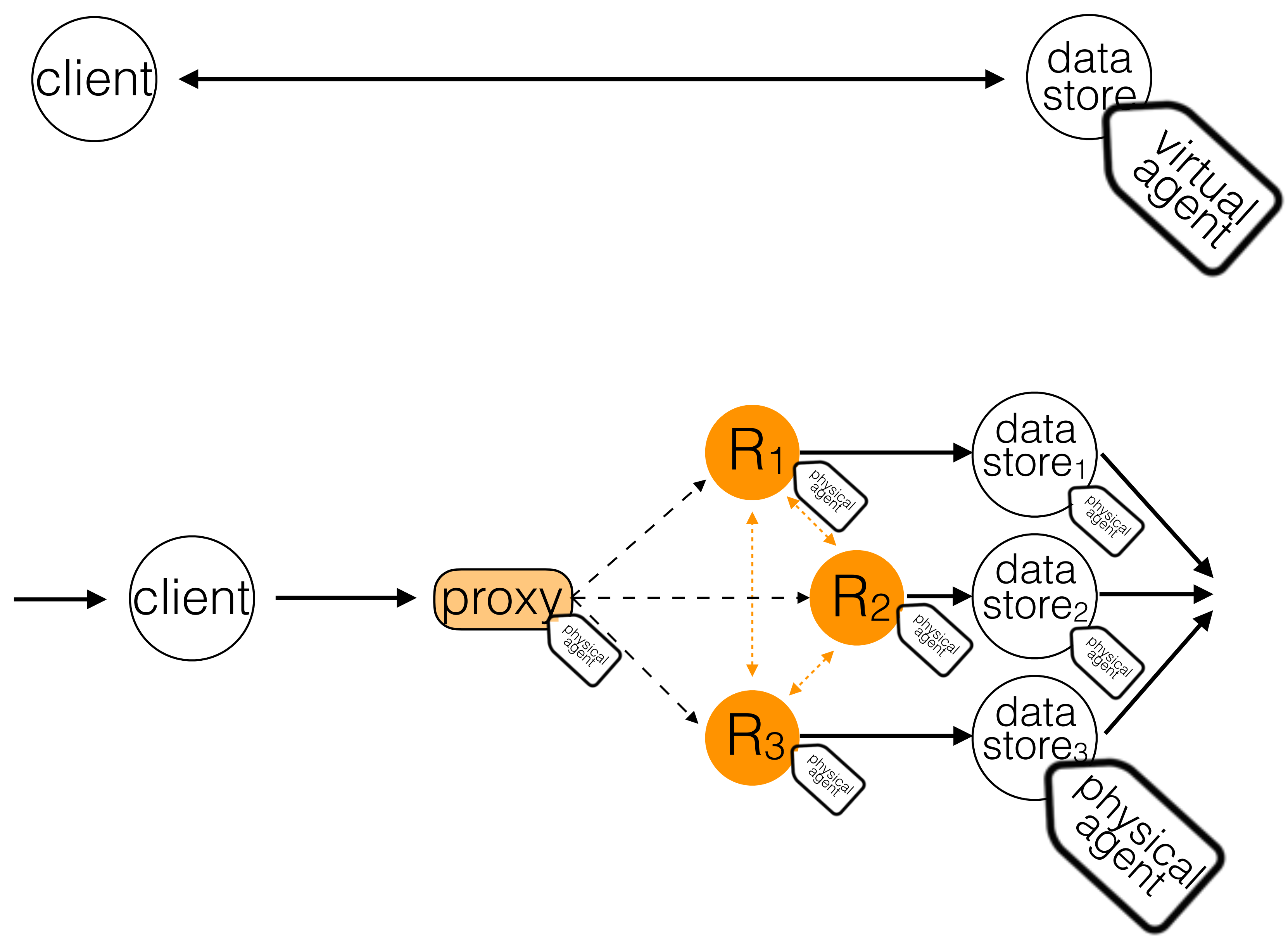


replicate(datastore,3) = proxy + R*3 + datastore*3

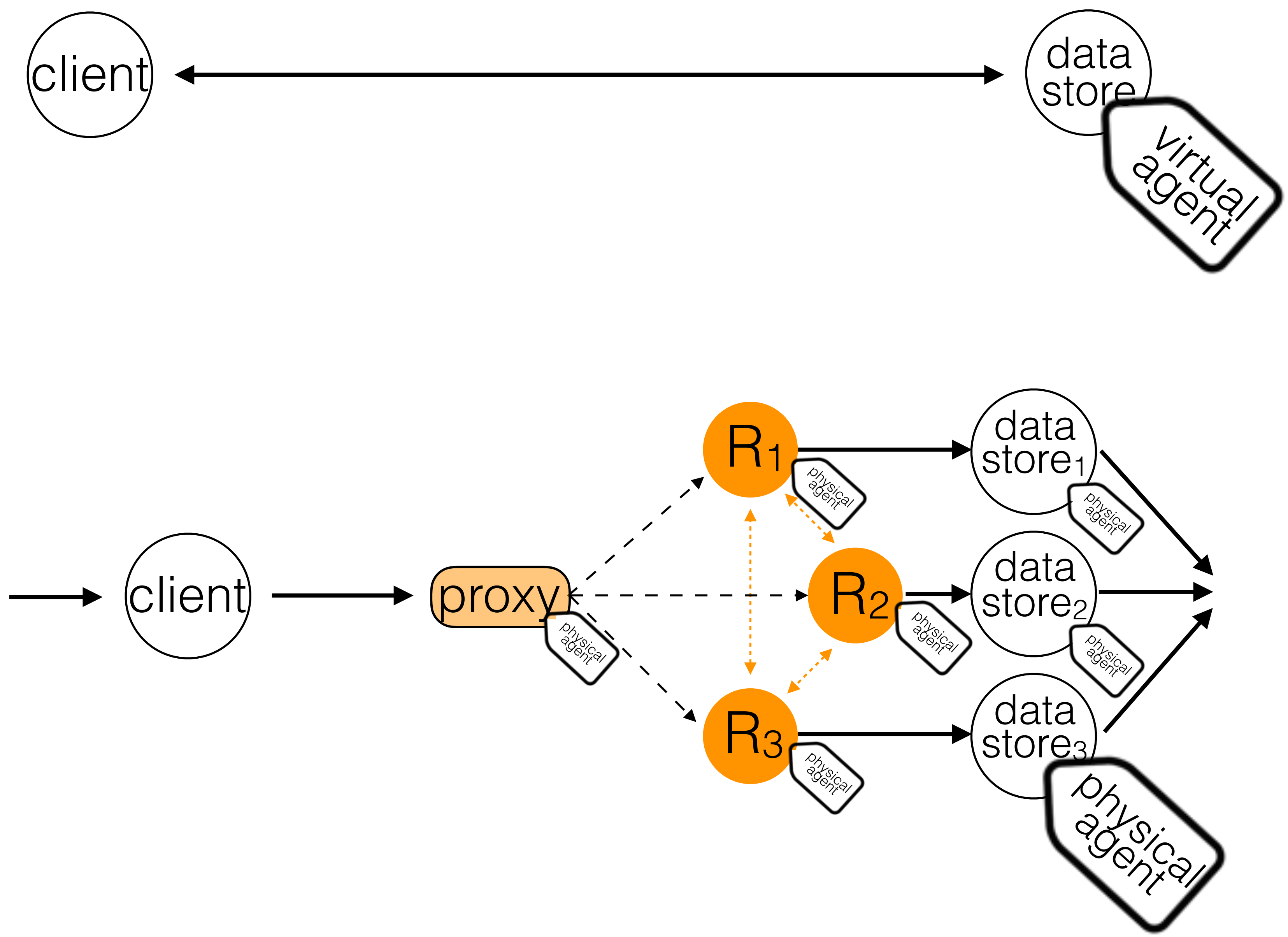
How do we know that a transformation did not break an agent?



How do we know that a transformation did not break an agent?

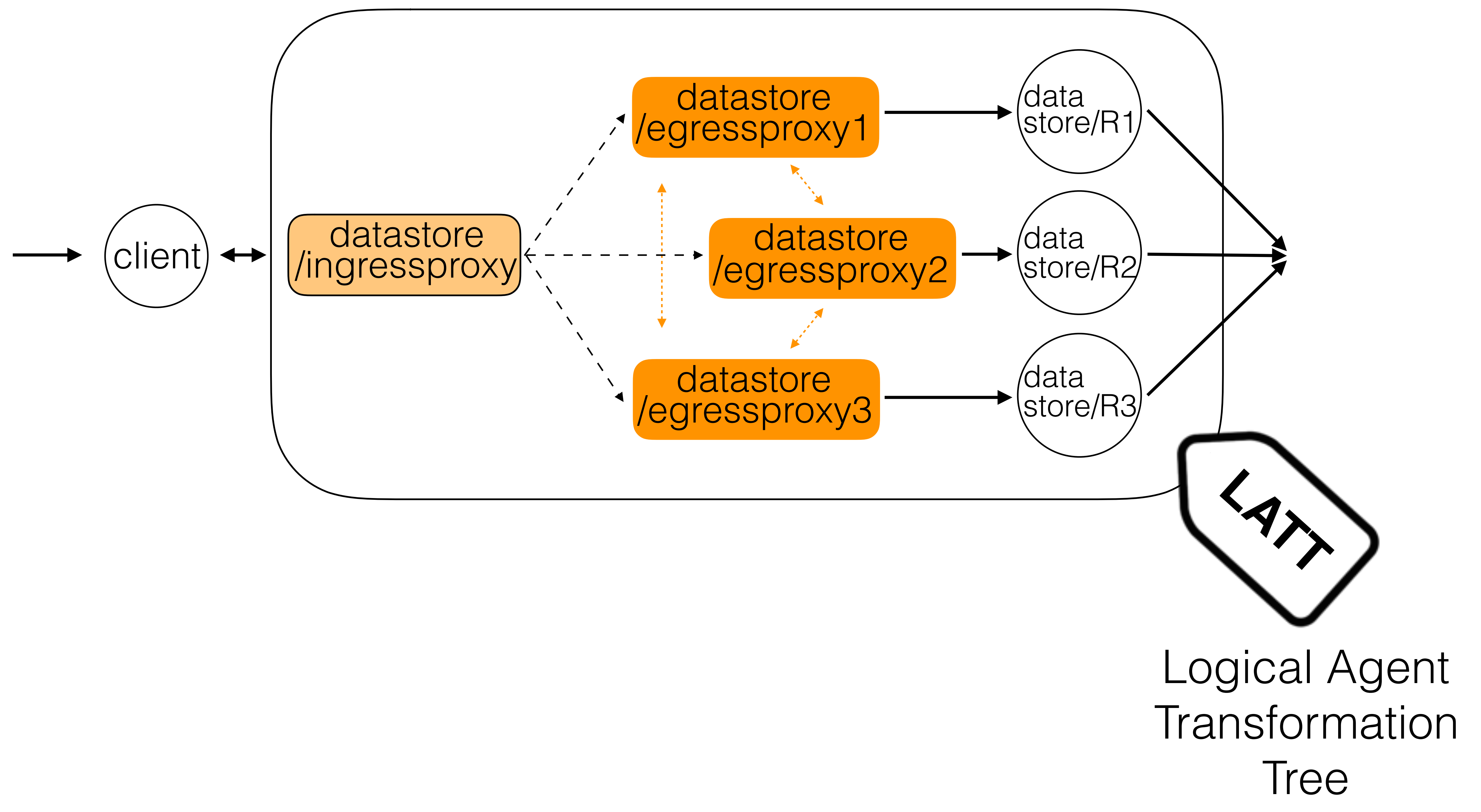


Use refinement mappings to prove equivalence




client
data
store

Use path names to represent transformations



Supported transformations

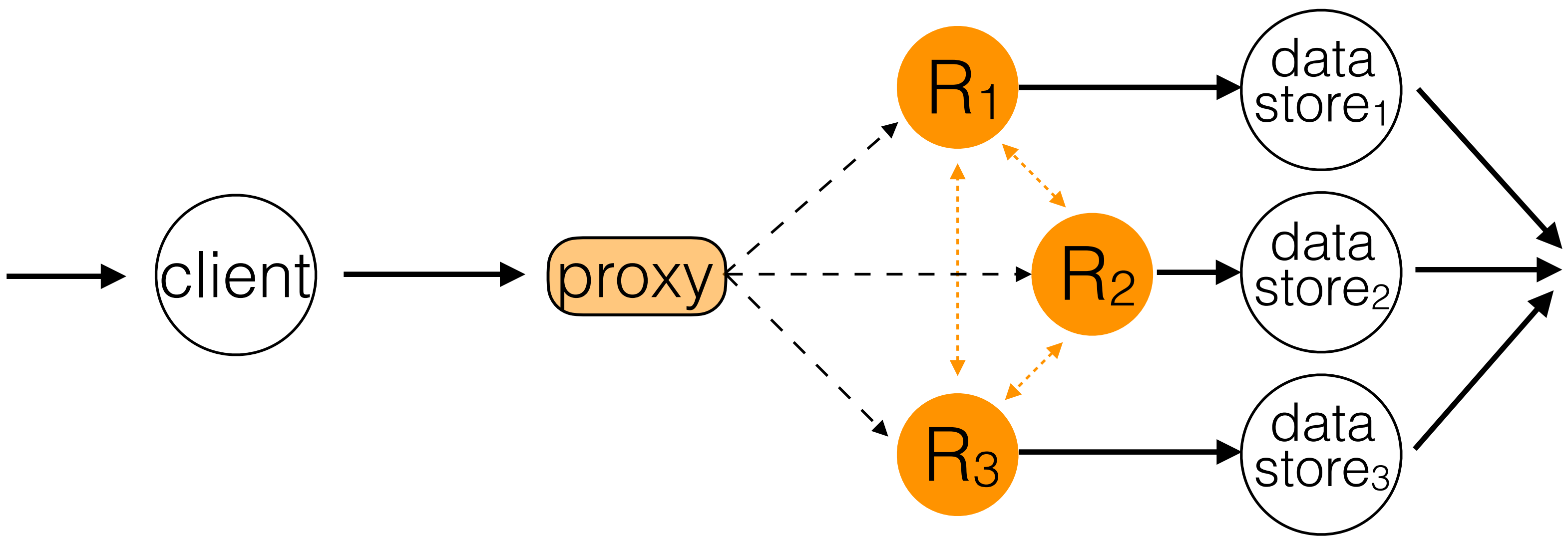
- replication
- byzantine resistance
- batching
- sharding
- encryption/decryption
- compression
- load-balancing
- deduplication

Approach

Create a theoretical model using abstractions

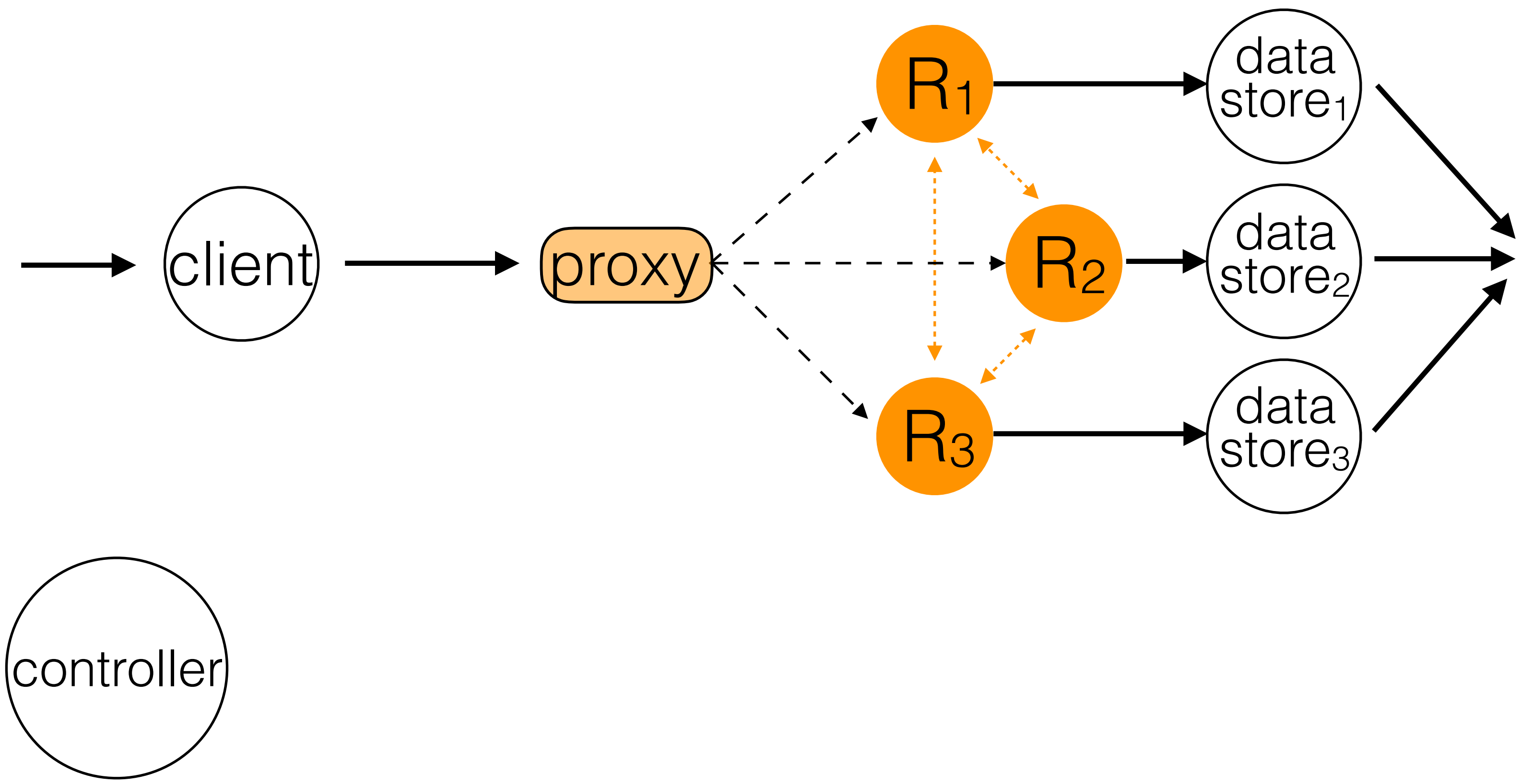
Deploy the distributed system using this model

Deployment



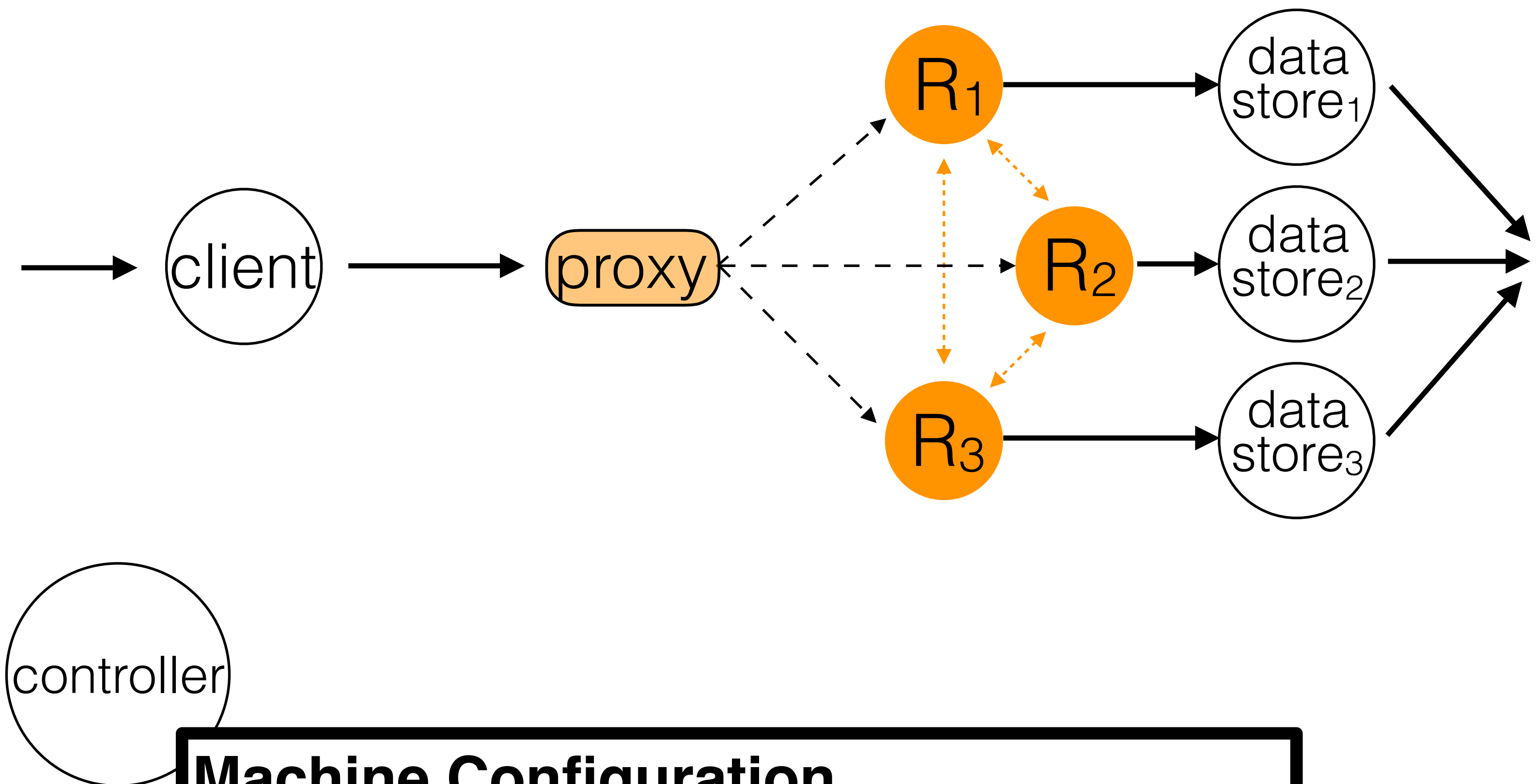
- Create a configuration from the model
- Deploy processes on boxes depending on the configuration

Deployment



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

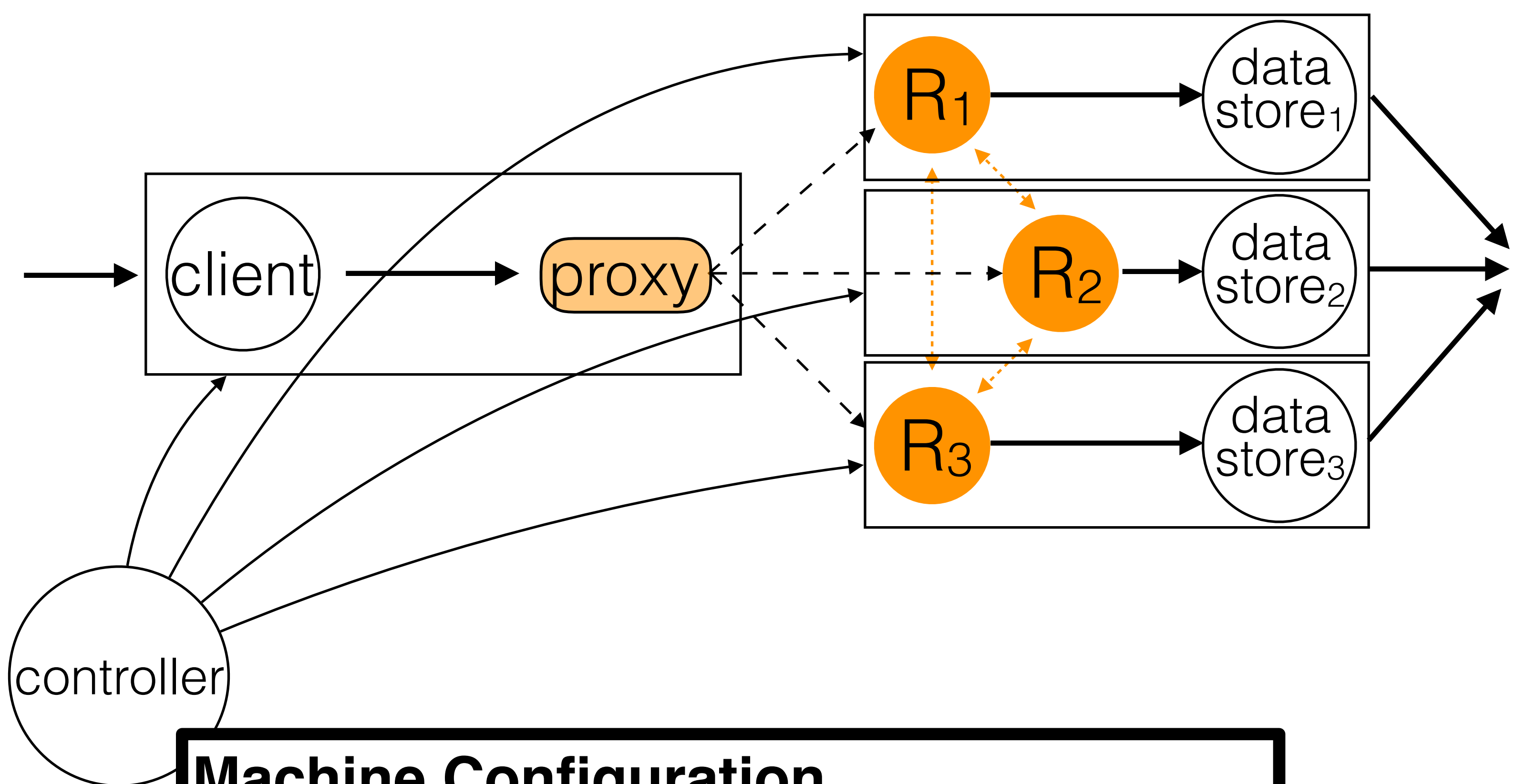
Deployment



Machine Configuration

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

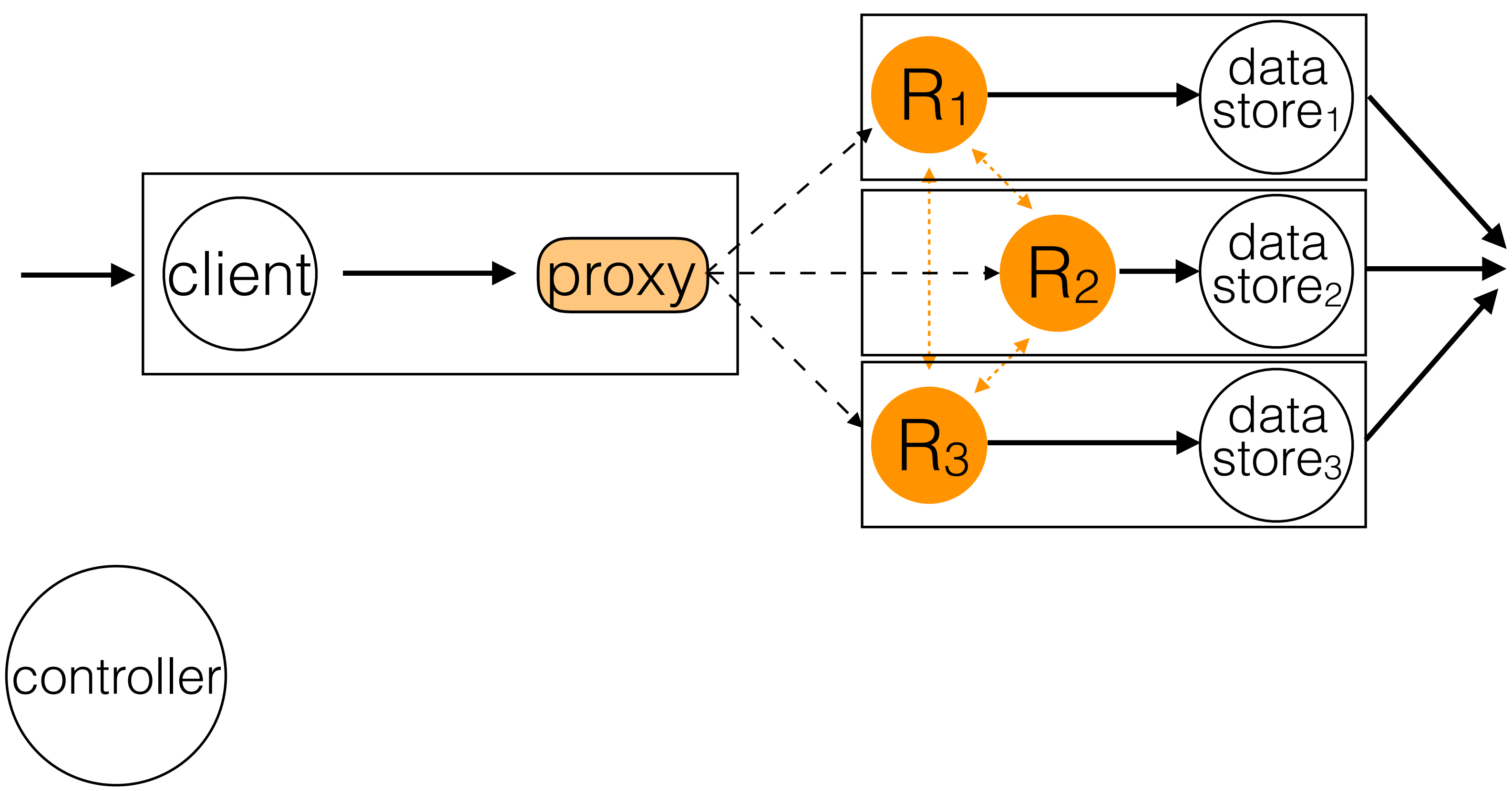
Deployment



<u>Machine Configuration</u>

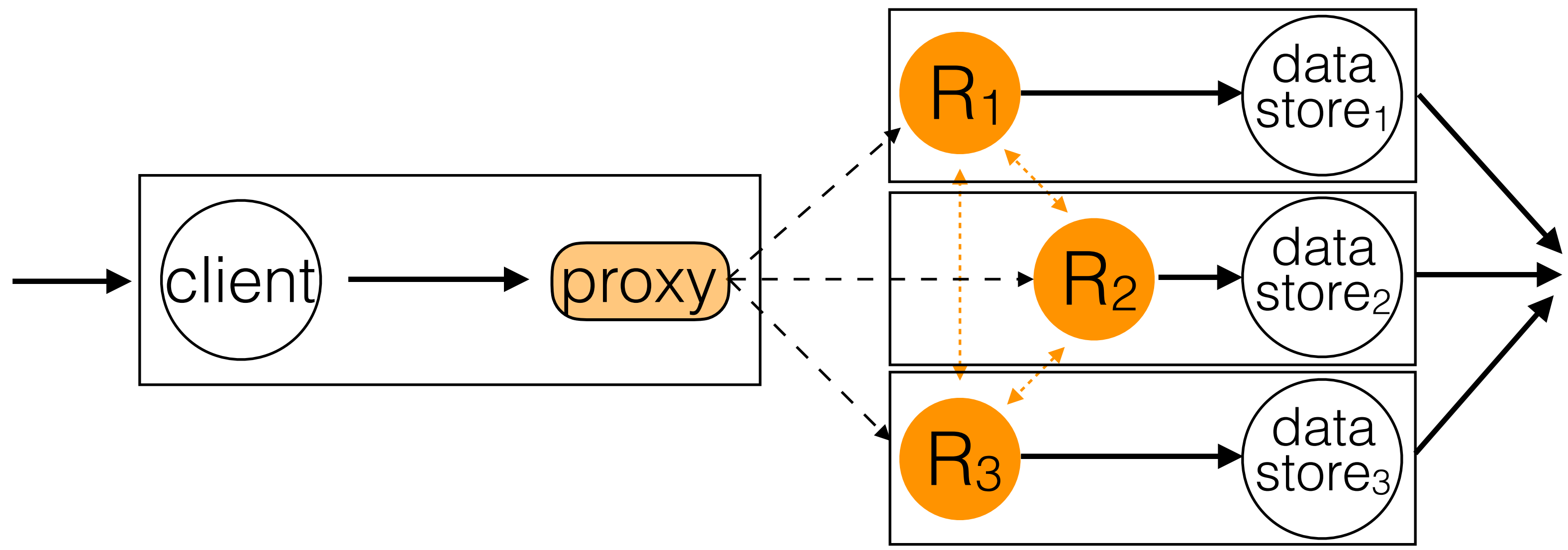
client 192.168.7.56 client.cpp
proxy 192.168.7.56 proxy.cpp
R1 192.168.7.80 replica.cpp
R2 192.168.7.81 replica.cpp
R3 192.168.7.82 replica.cpp
data store1 192.168.7.80 datastore.cpp
data store2 192.168.7.81 datastore.cpp
data store3 192.168.7.82 datastore.cpp

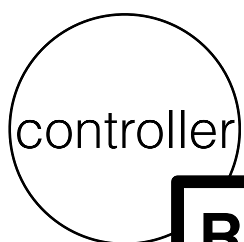
Dynamic Routing



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

Dynamic Routing





<u>Routing Configuration</u>

client datastore proxy

proxy datastore R1,R2,R3

R1 datastore datastore1

R2 datastore datastore2

R3 datastore datastore3

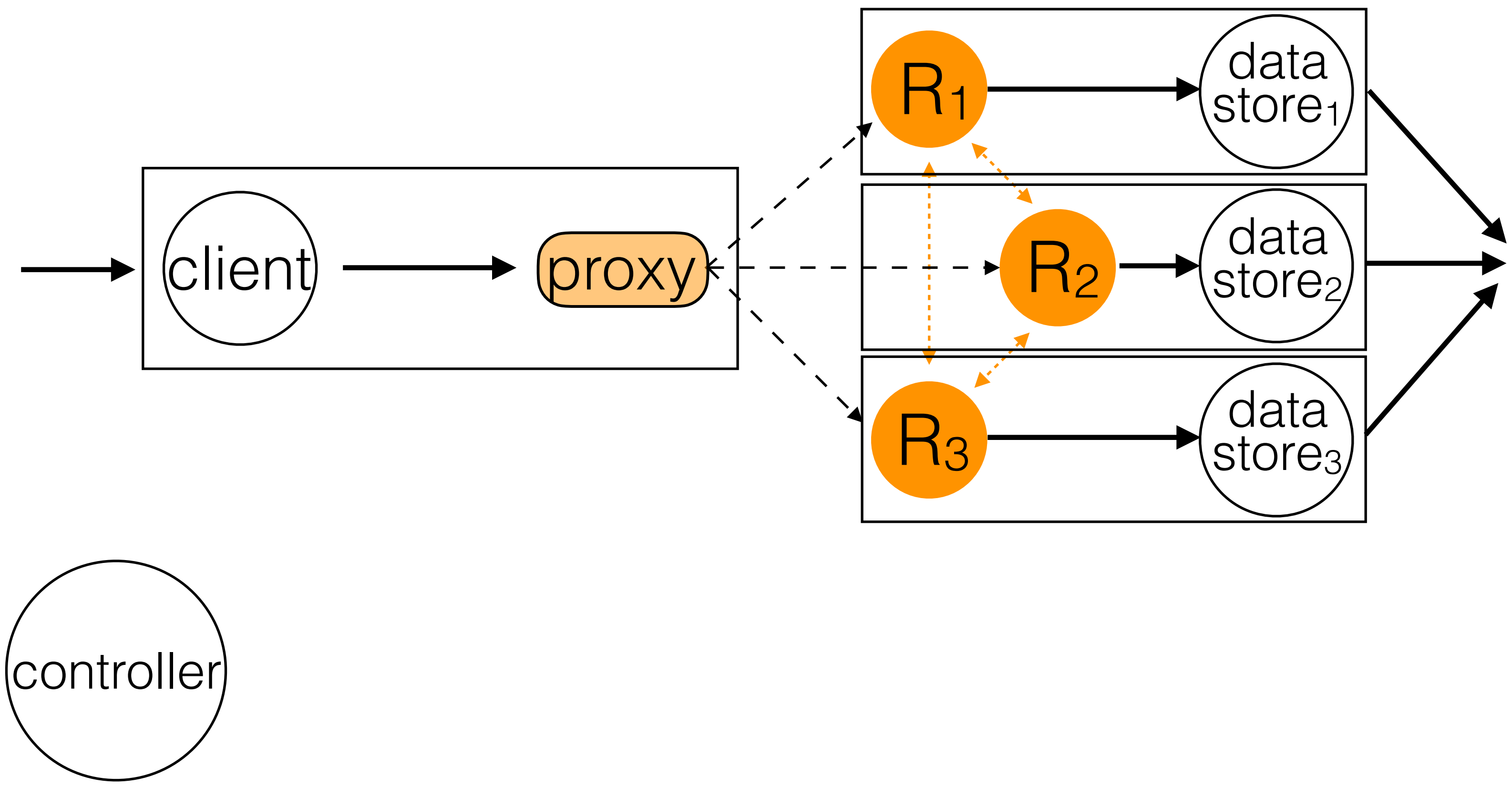
datastore1 client client

datastore2 client client

datastore3 client client

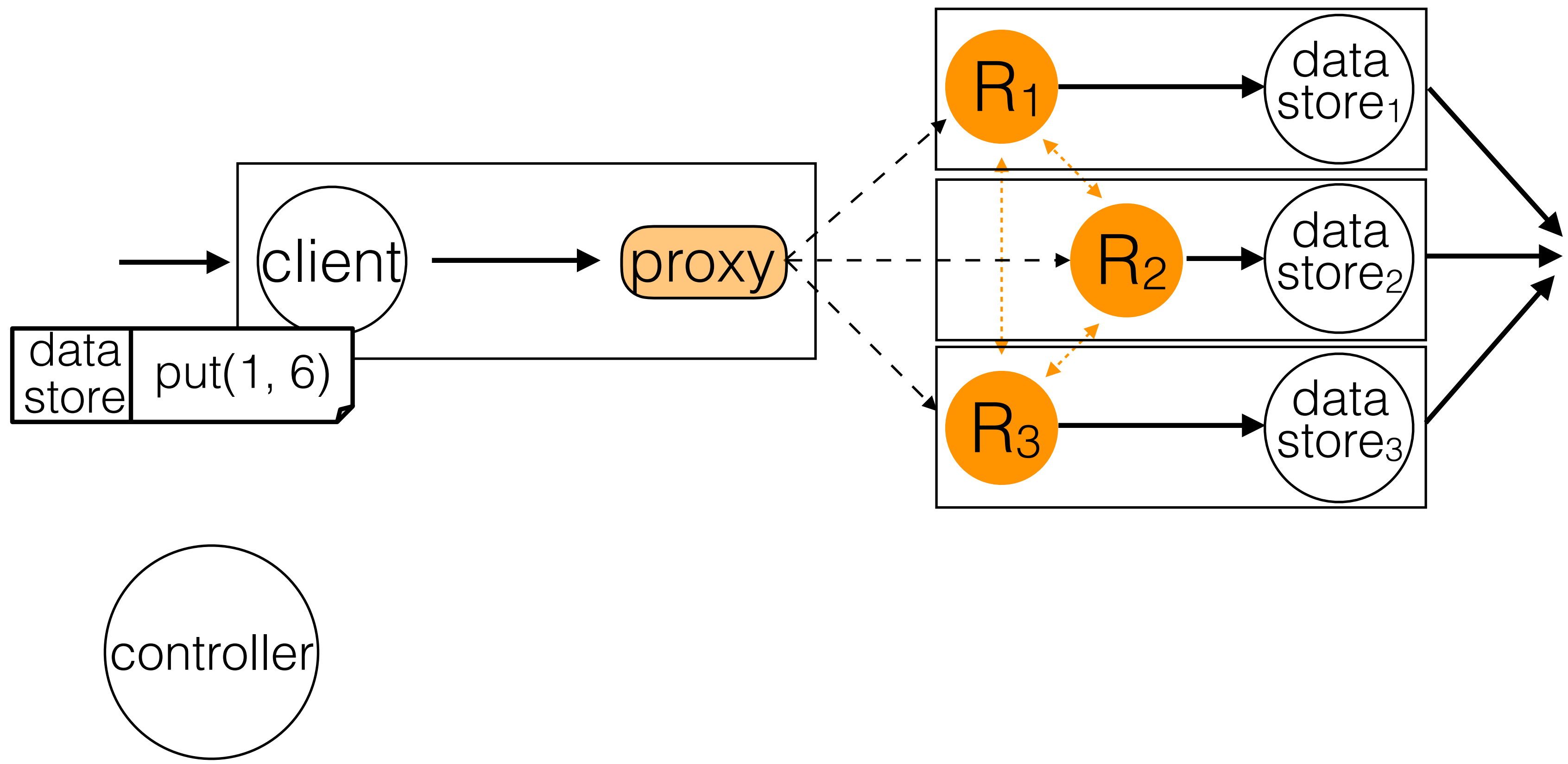
Dynamic Routing

msg:



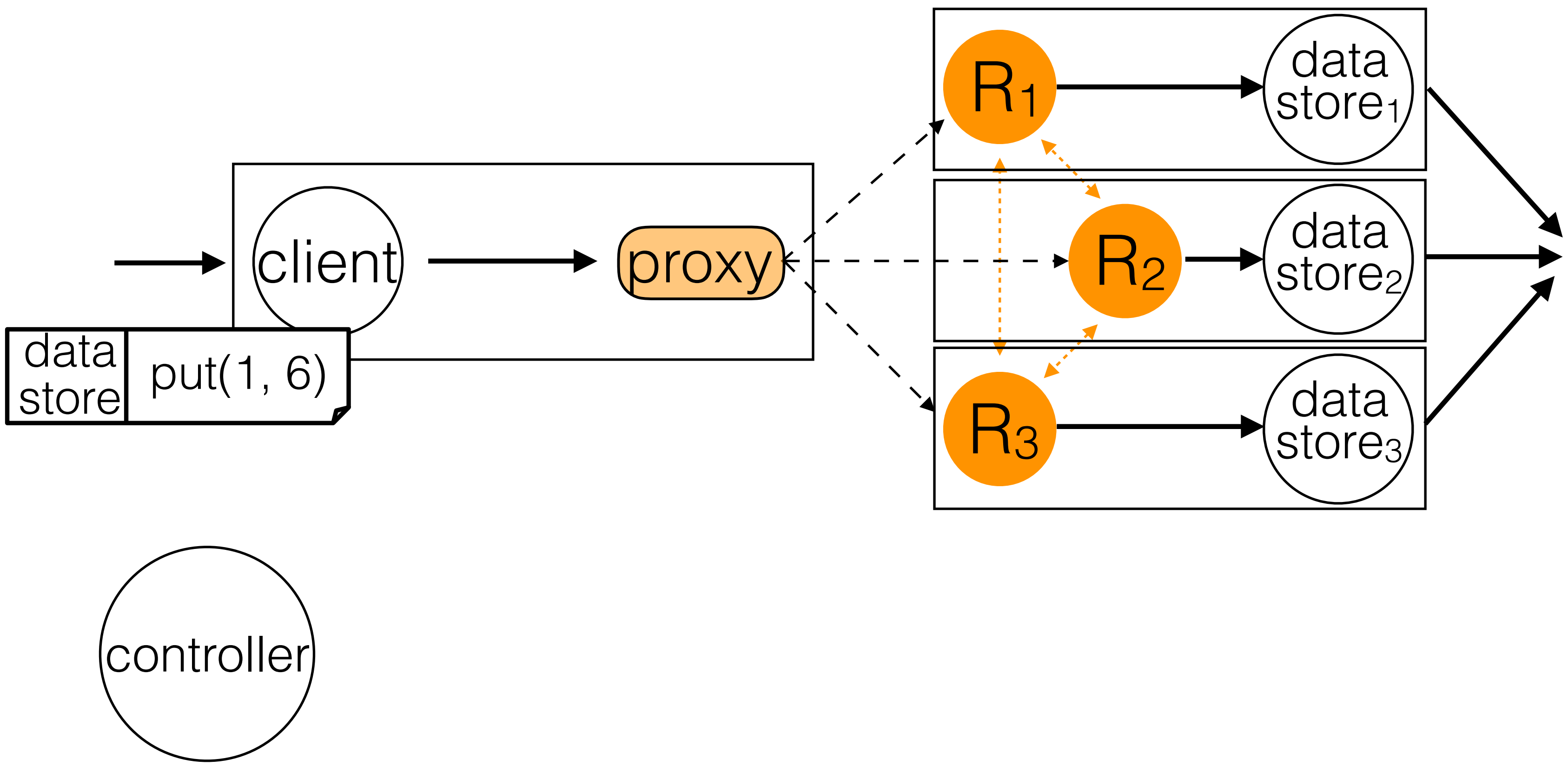
Dynamic Routing

msg: | dest | payload |



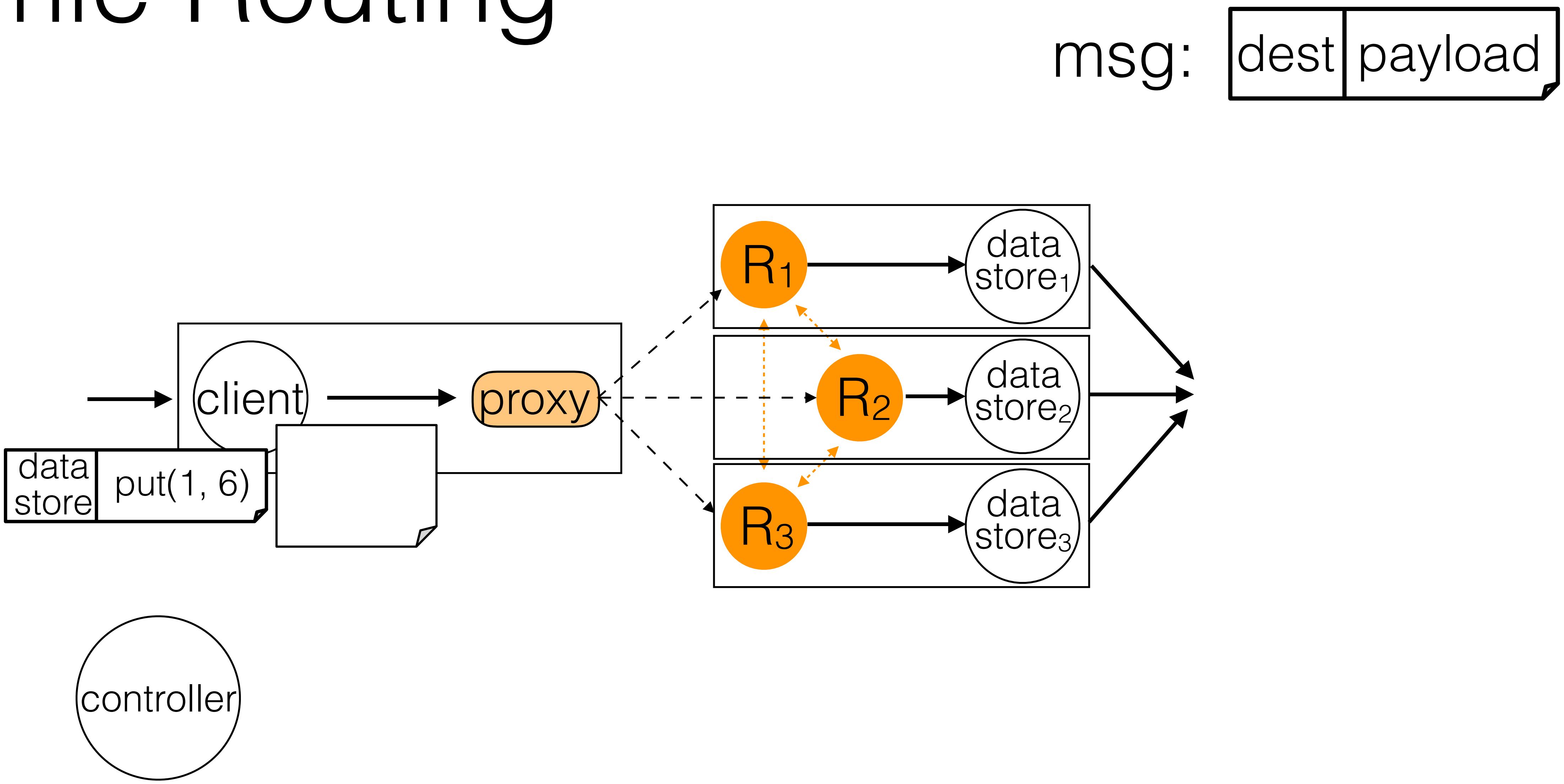
Dynamic Routing

msg:



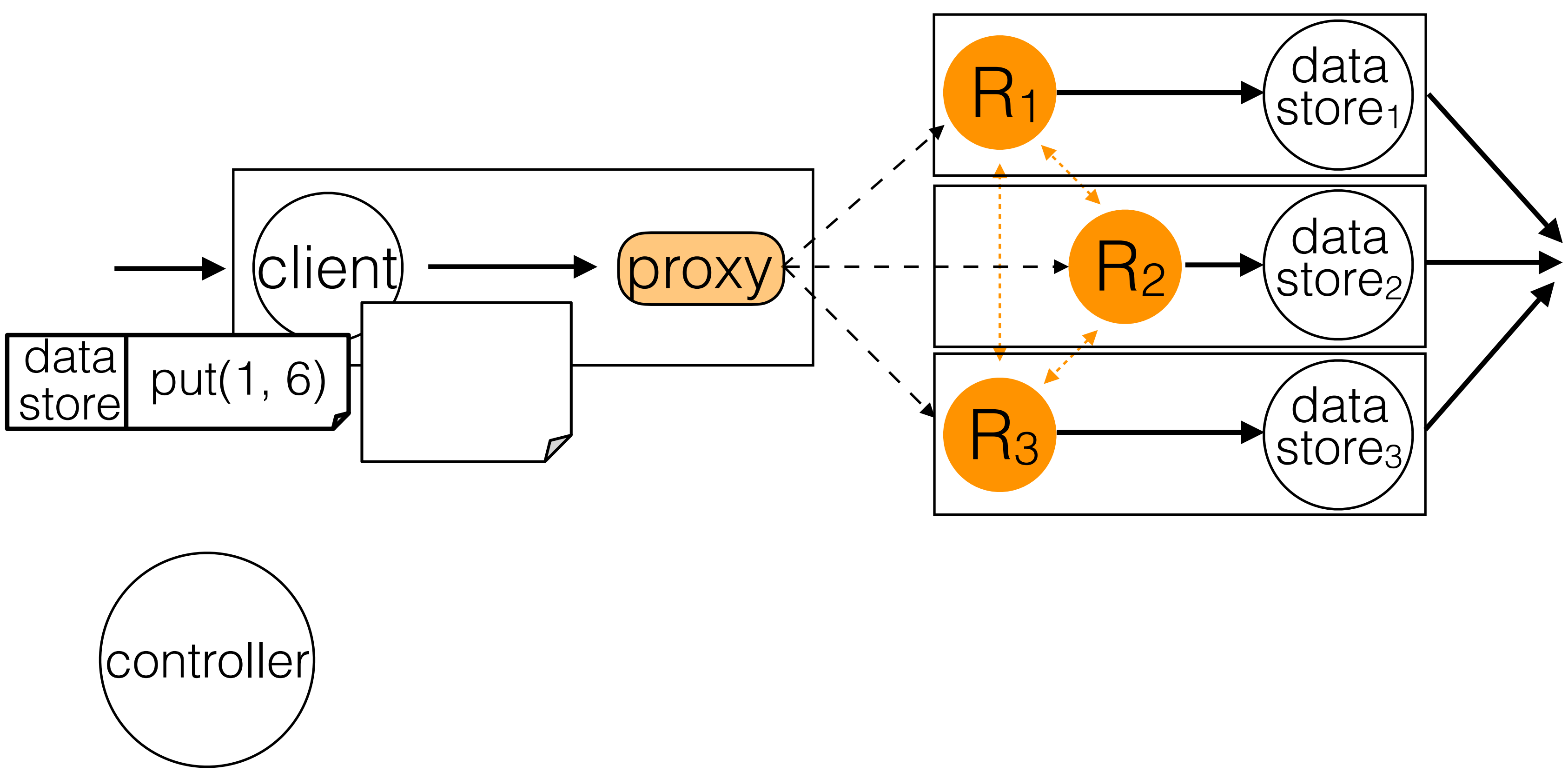
- send msg(dest,payload)

Dynamic Routing



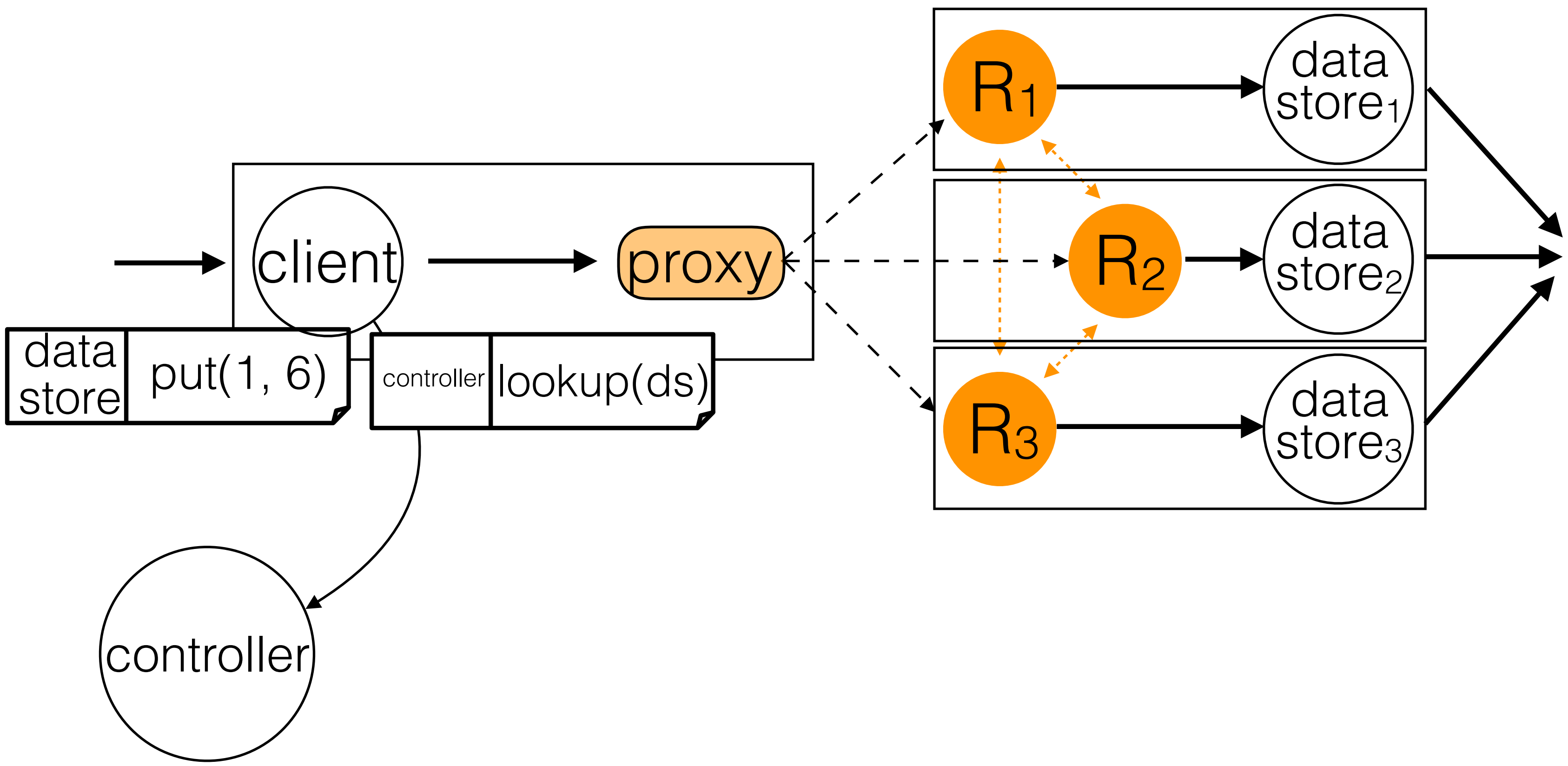
- send msg(dest,payload)
- look for dest in routing table

Dynamic Routing



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

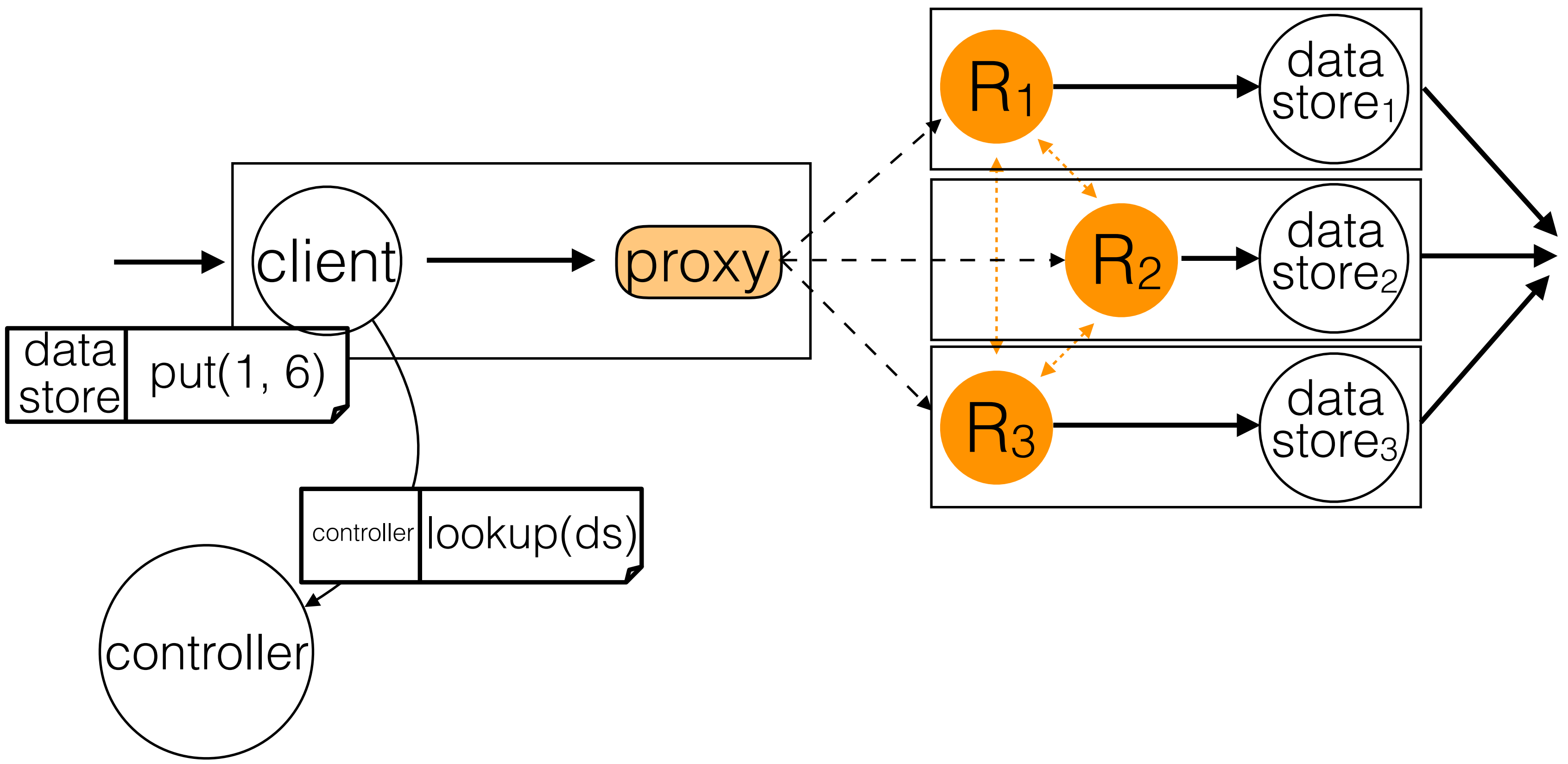
Dynamic Routing



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

Dynamic Routing

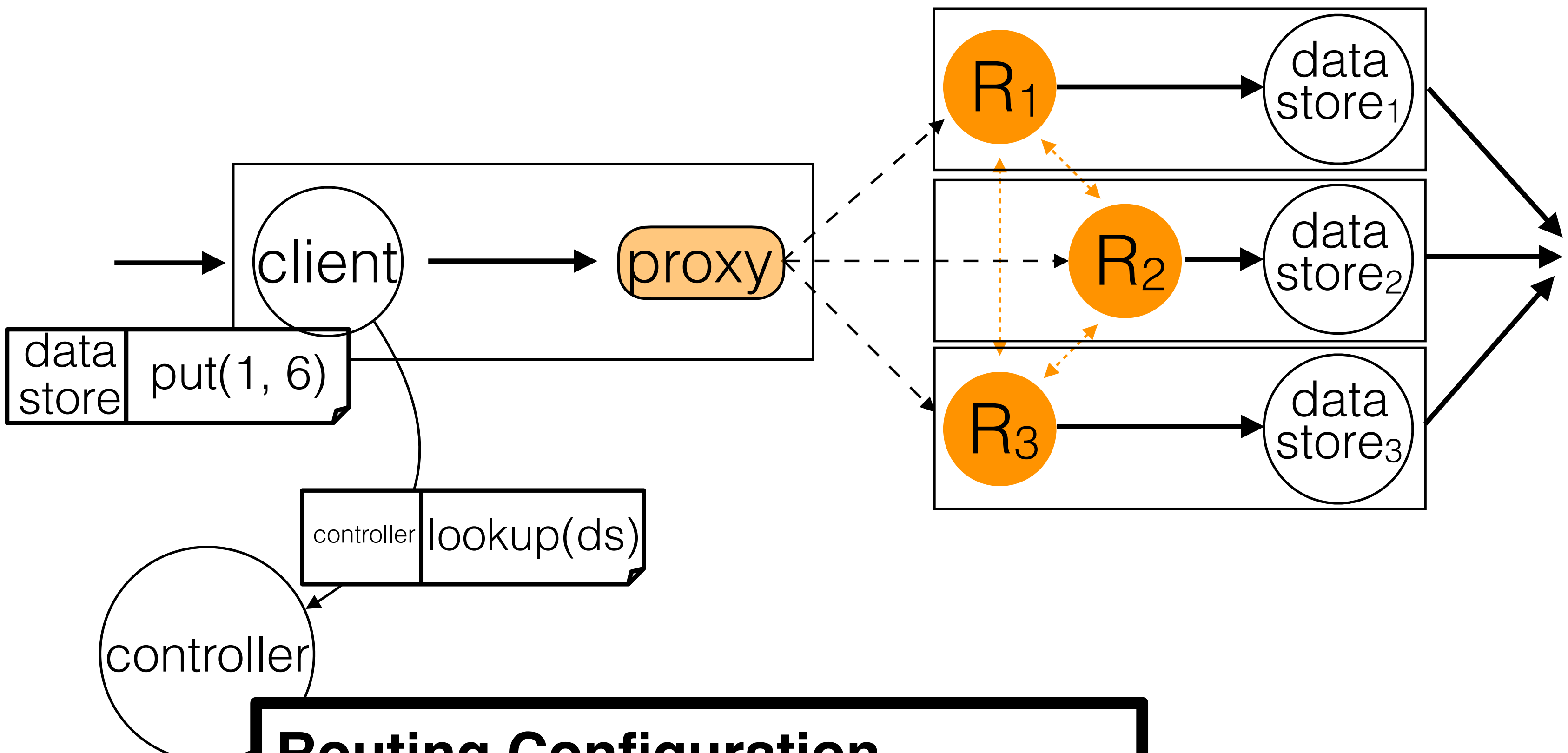
msg: 



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

Dynamic Routing

msg: dest | payload



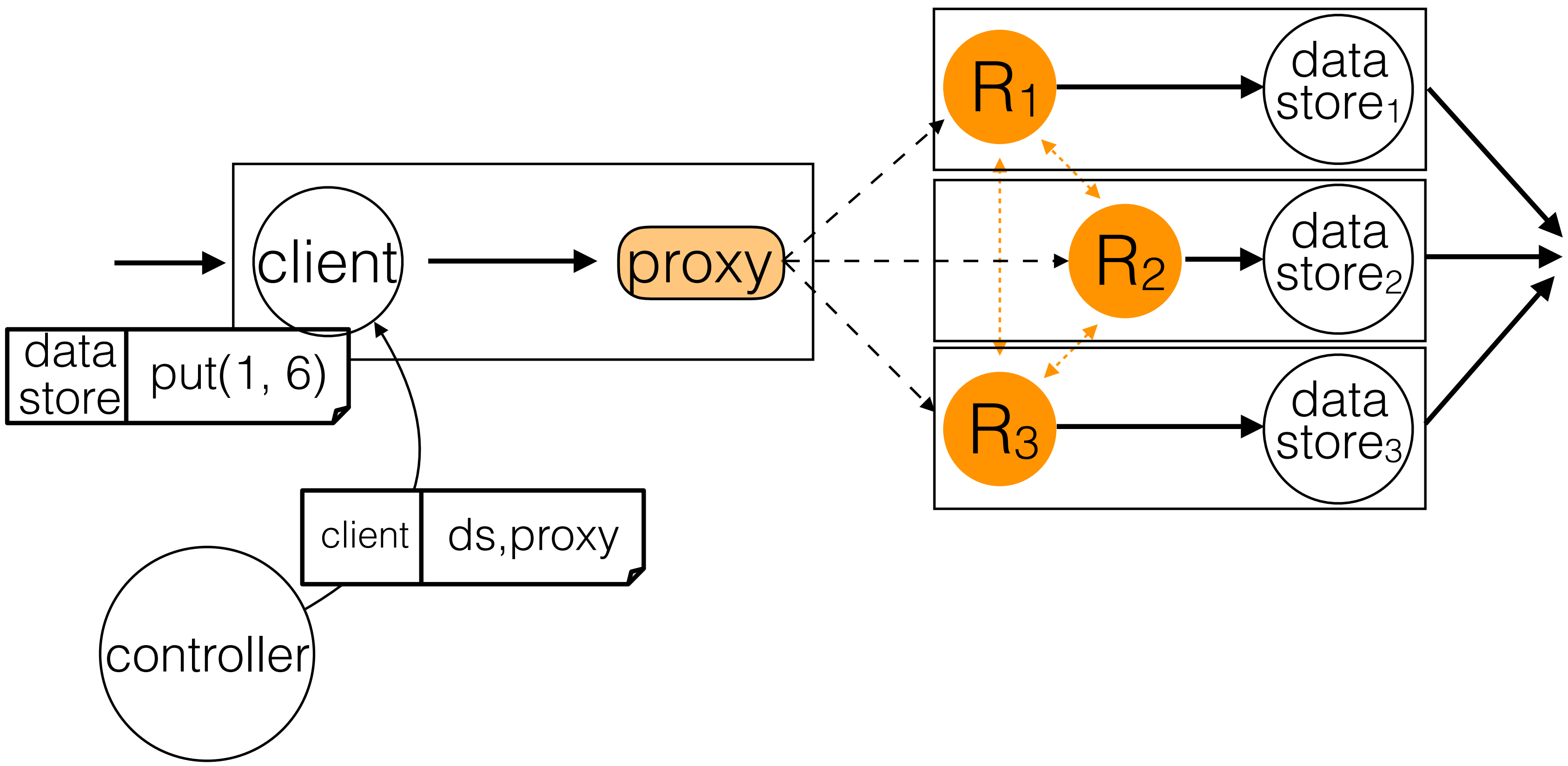
- send msg(dest,pay
- look for dest in rout
- dest not present
- send lookup messa

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

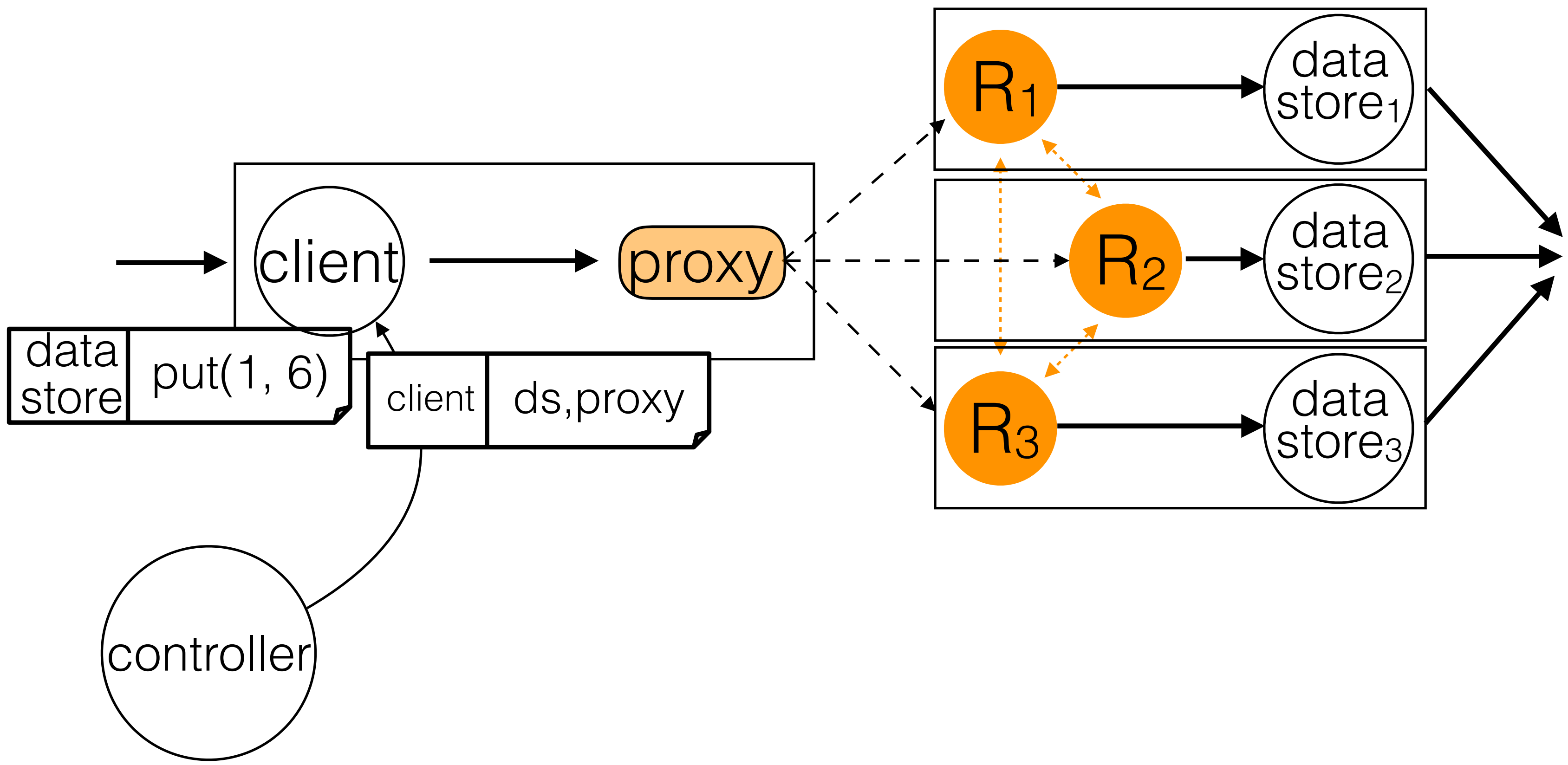
msg: | dest | payload |



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

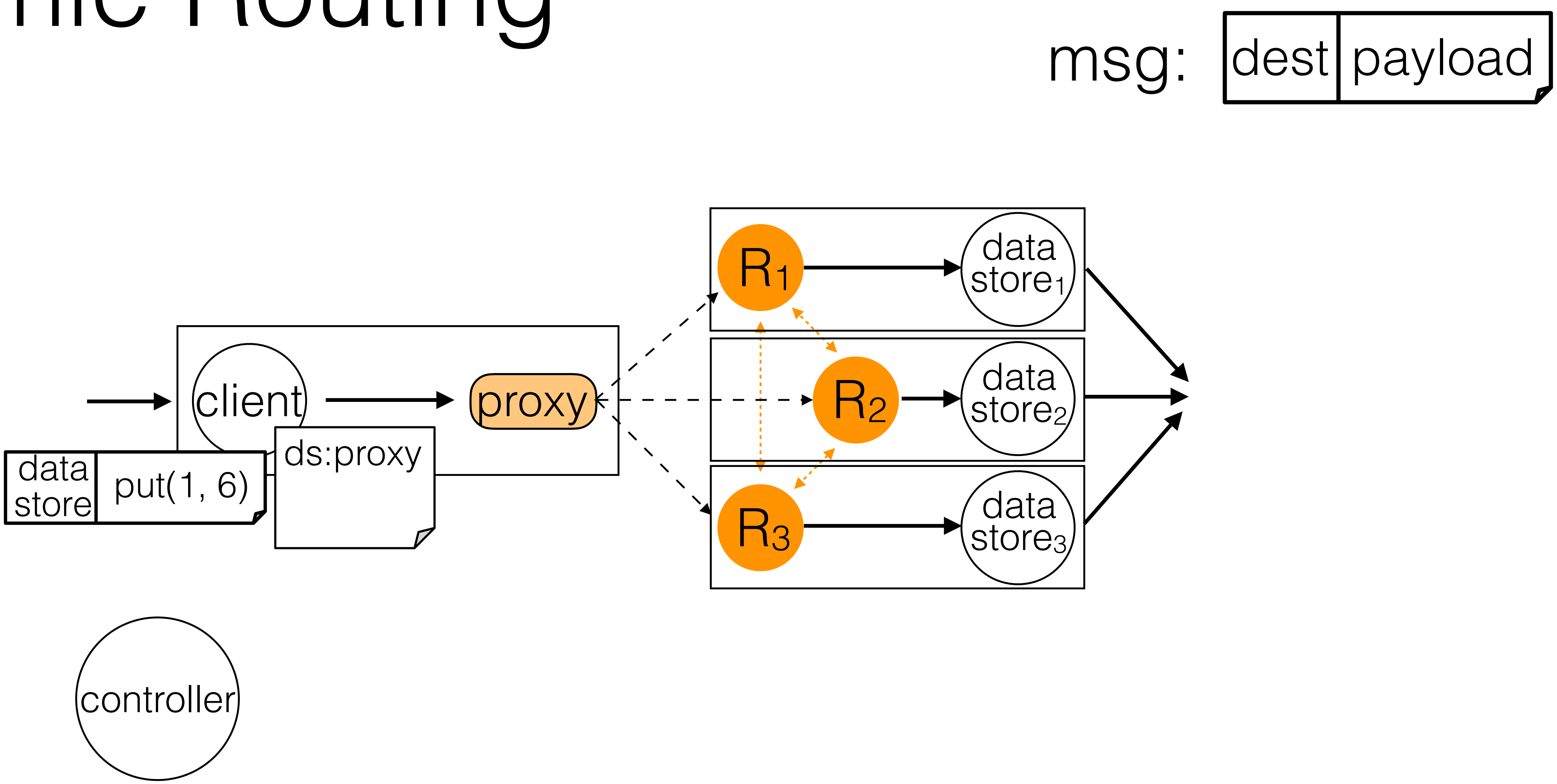
Dynamic Routing

msg:



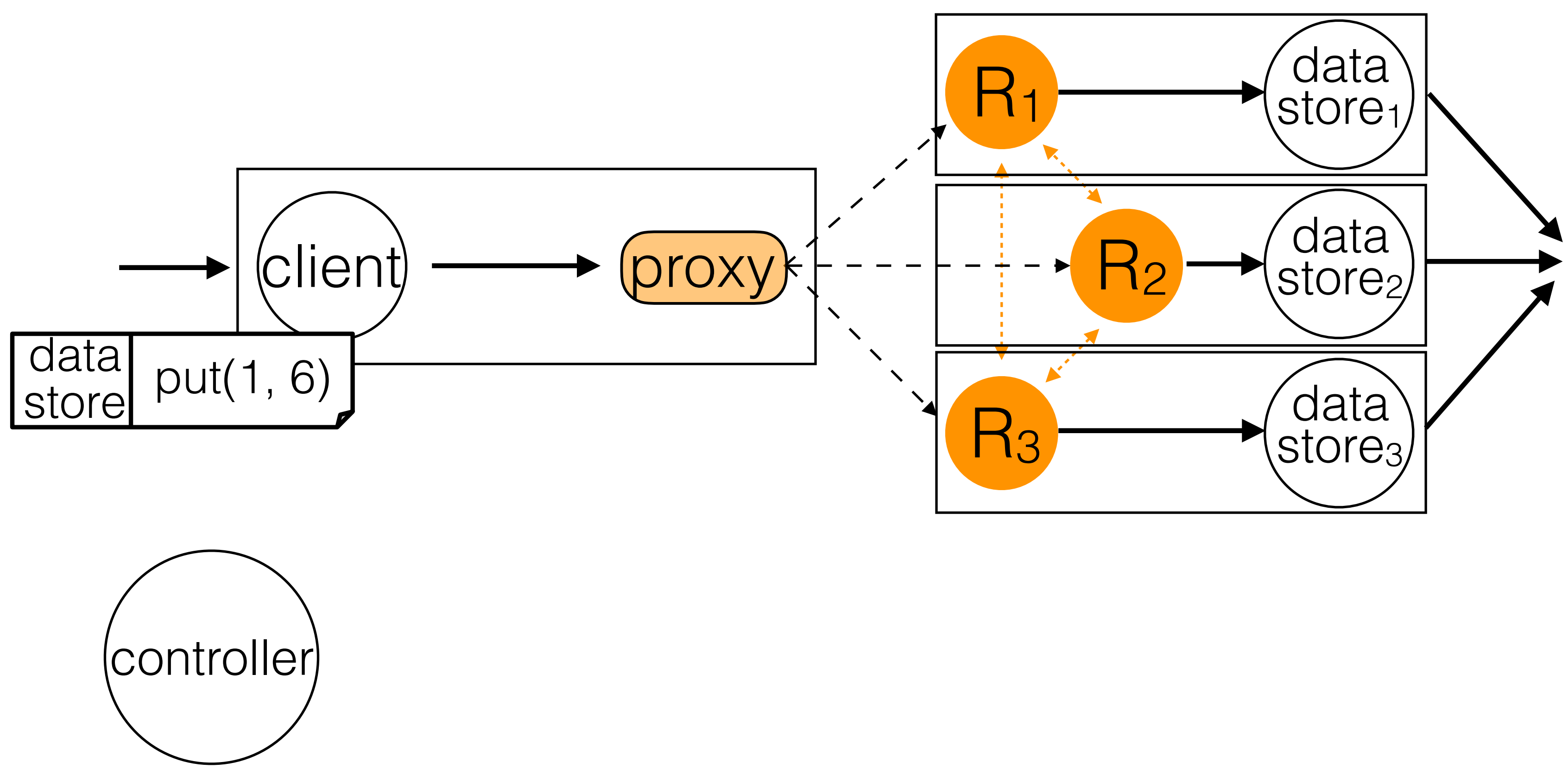
- send msg(dest,payload)
- look for dest in routing table
- dest not present
- send lookup message to controller
- get route mapping

Dynamic Routing



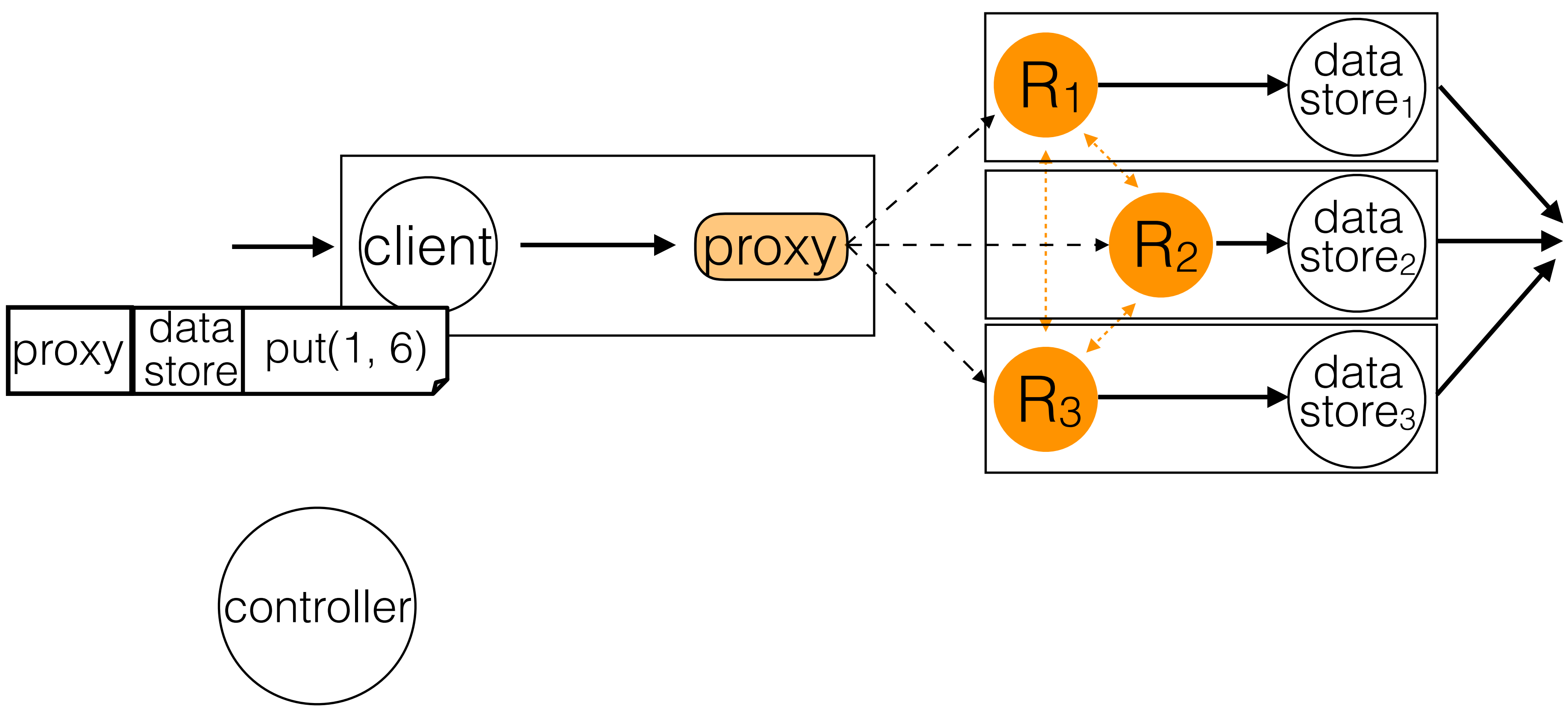
- send msg(dest,payload)
- look for dest in routing table
- dest not present
- send lookup message to controller
- get route mapping
- update routing table

Dynamic Routing



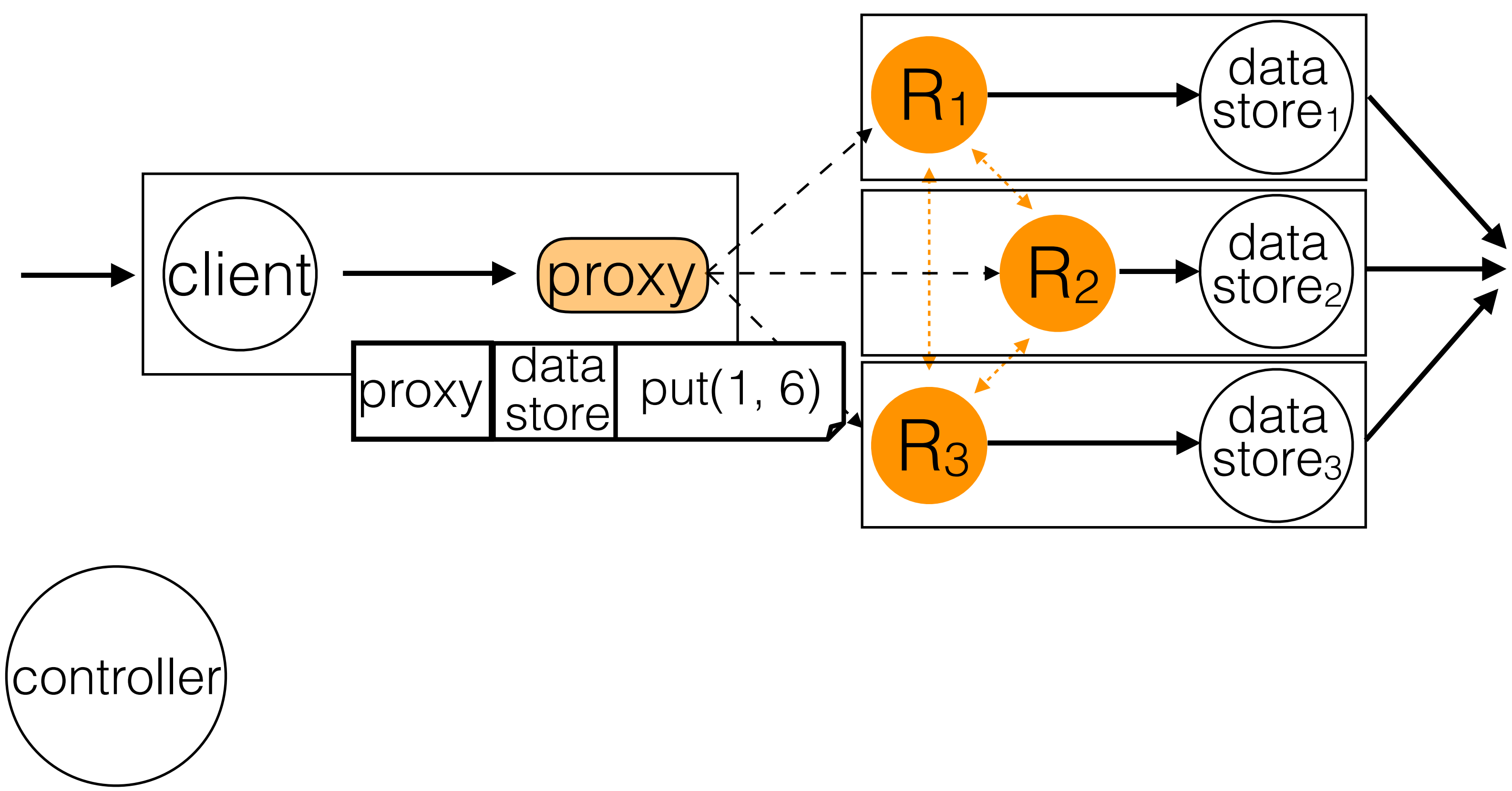
- send msg(dest,payload)
- look for dest in routing table
- dest not present
- send lookup message to controller
- get route mapping
- update routing table
- send msg(dest,payload)

Dynamic Routing



- 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.