Cost-optimized, Policy-based Data Management in Cloud Environments

Ilir Fetai
Filip-Martin Brinkmann

Databases and Information Systems Research Group
University of Basel

Current State in the Cloud: A “zoo” of different DBMS

- Current DBMS differ in:
  - APIs
  - Guarantees (Ex: consistency models, replication, etc.)
  - Limitations (Ex: no support for archiving, etc.)
  - Costs

DMS “wish list”

- DMS = Data Management System
  - Construct “my” DBMS via *declarative specifications*
  - **Declarative specifications = Policies**

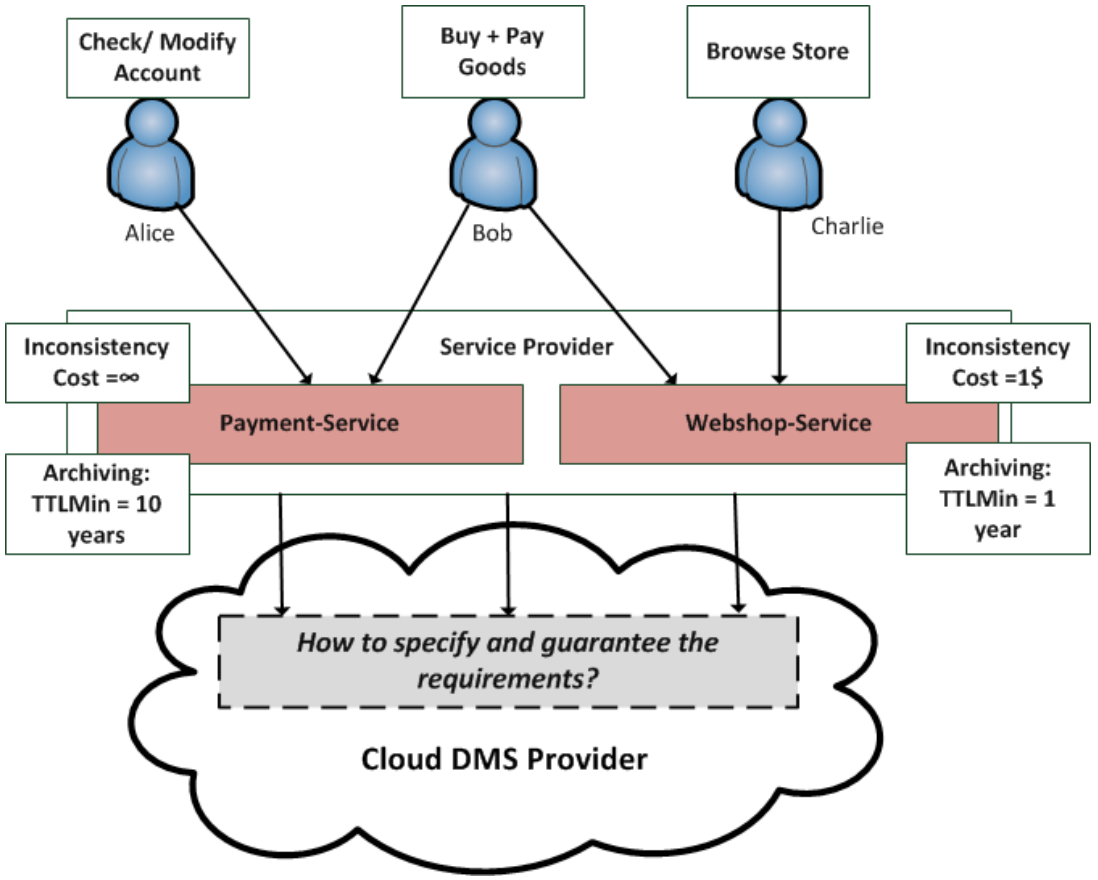
- Policy management needed
  - A way to express requirements of applications, transactions, end-users, etc.,

    and monitor their fulfilment
  - Examples:
    - Control **data consistency**
    - Specify possible **data replication locations**
    - Control **data archiving**
    - Control **costs!**

- Modular DMS architecture needed

Scenario

- **Client:**
  - use services
  - have different requirements
- **Service Provider:**
  - provides different services
  - uses underlying DMS infrastructure
  - has requirements on data management
- **Cloud DMS Provider:**
  - provides the DMS infrastructure



Agenda

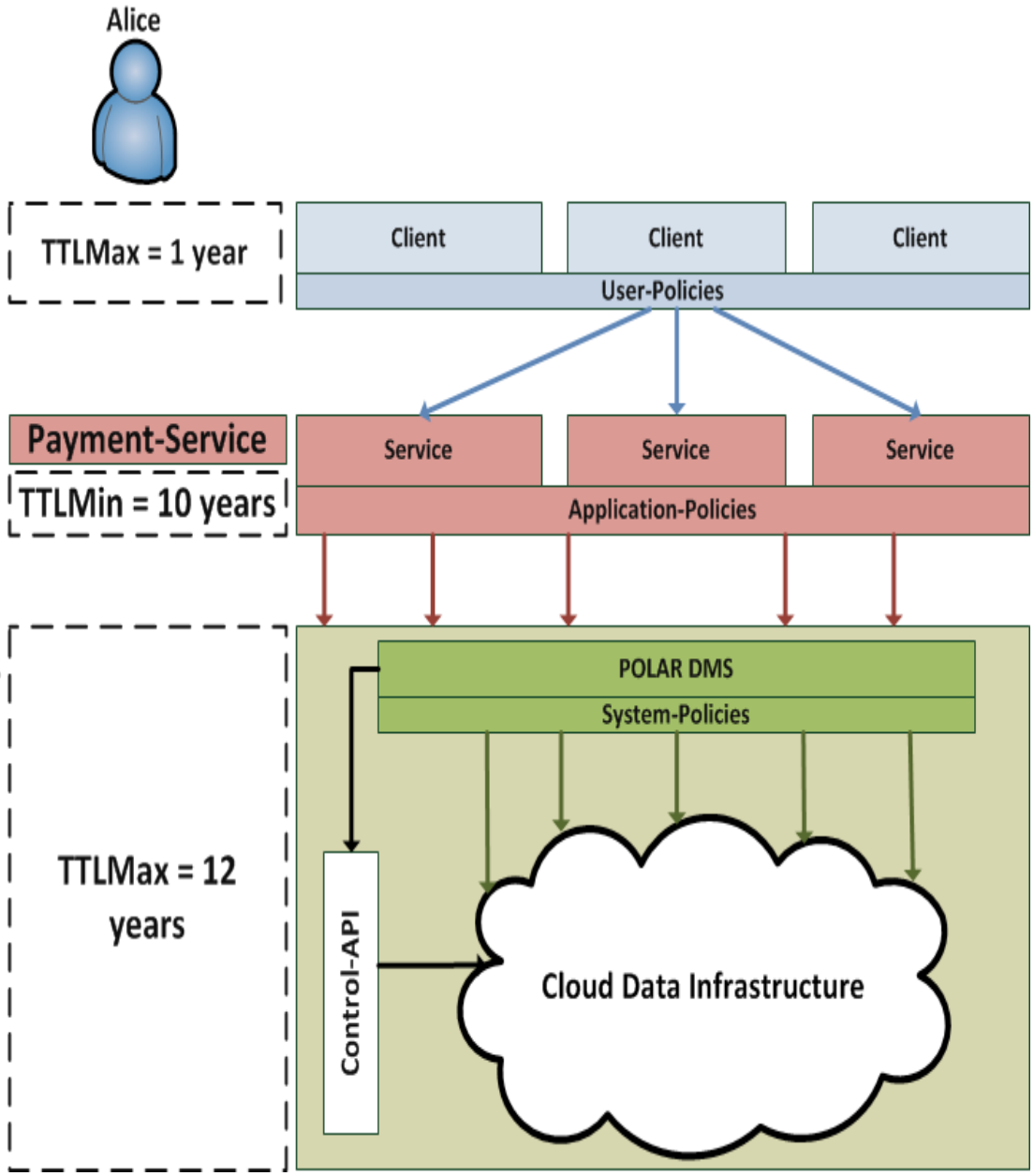
- Policy-based and modular Data Management Systems (POLAR DMS)
- Cost-effective & Policy-based Data Management
  - Data Consistency
  - Data Replication
  - Data Archiving
- Architecture of POLAR DMS
- Conclusion

Agenda

- Policy-based and modular Data Management Systems (POLAR DMS)
- Cost-effective & Policy-based Data Management
  - Data Consistency
  - Data Replication
  - Data Archiving
- Architecture of POLAR DMS
- Conclusion

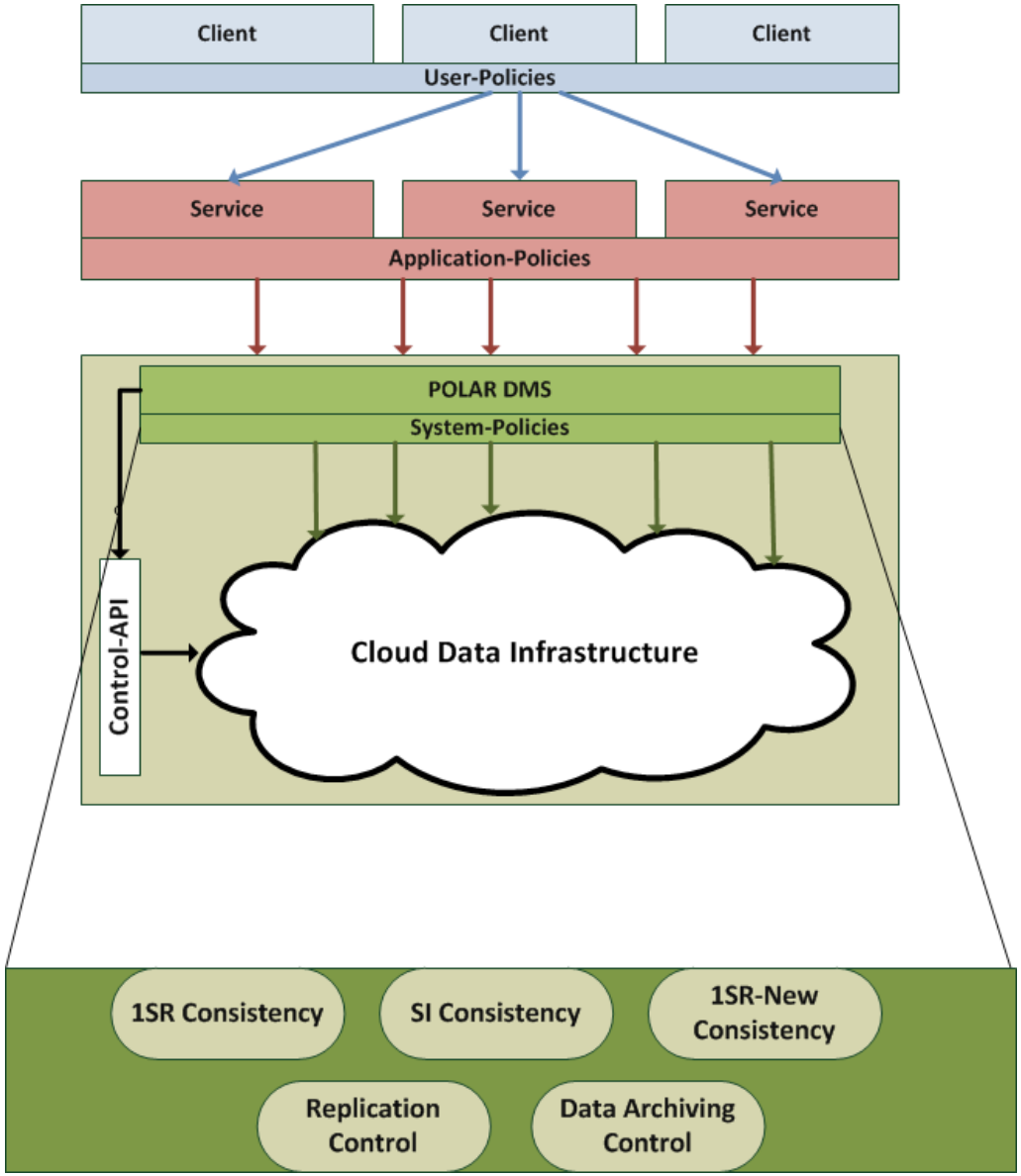
POLAR DMS - Requirements

- Requirements: User-friendly & understandable policy-based and modular DMS
  - Specify policies at different levels: Cloud provider, application/service provider, but even end-customer
  - DMS adjusts at runtime based on policies
  - Minimal core
  - Implement DMS functionality as exchangeable modules



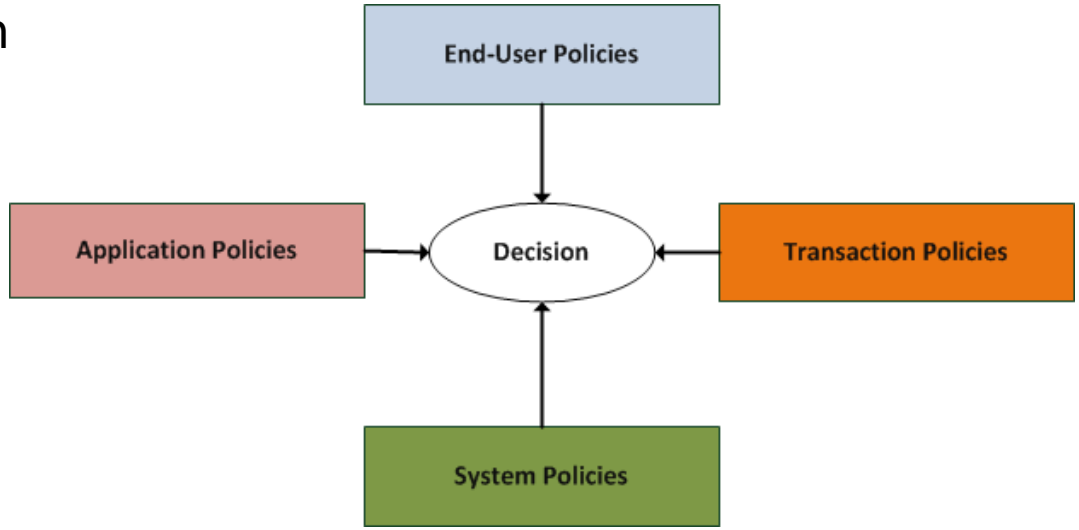
POLAR DMS - Requirements

- Requirements: User-friendly & understandable policy-based and modular DMS
  - Specify policies at different levels: Cloud provider, application/service provider, but even end-customer
  - DMS adjusts at runtime based on policies
  - Minimal core
  - Implement DMS functionality as exchangeable modules



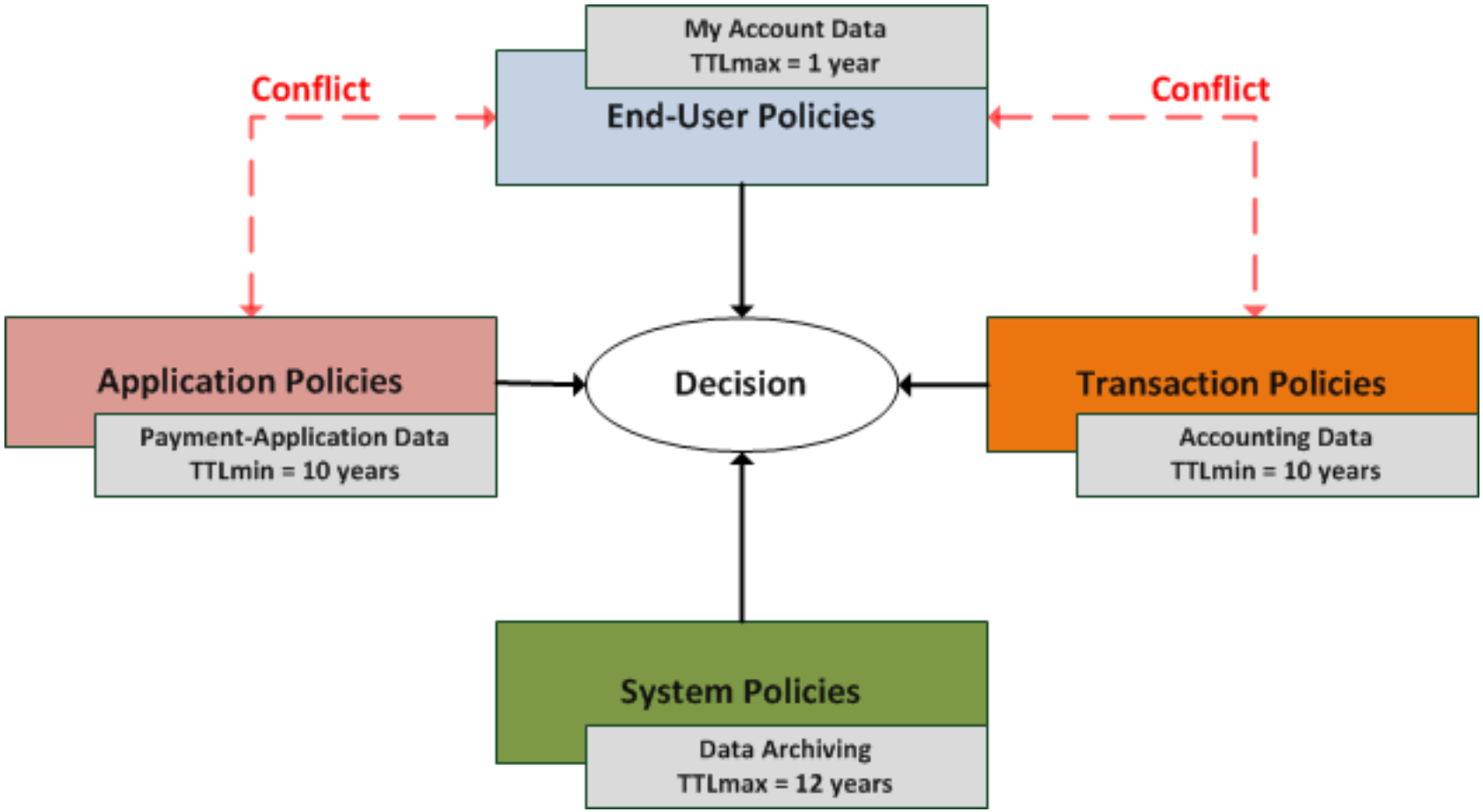
Policies in POLAR DMS

- End-user policies
  - May override default transaction policies
  - Ex: paid services
- Transaction-policies
  - May override default application policies
- Application-policies
  - Applications may specify default policies valid for entire application
- System-policies
  - The underlying system may also specify own policies



Conflicting Policies

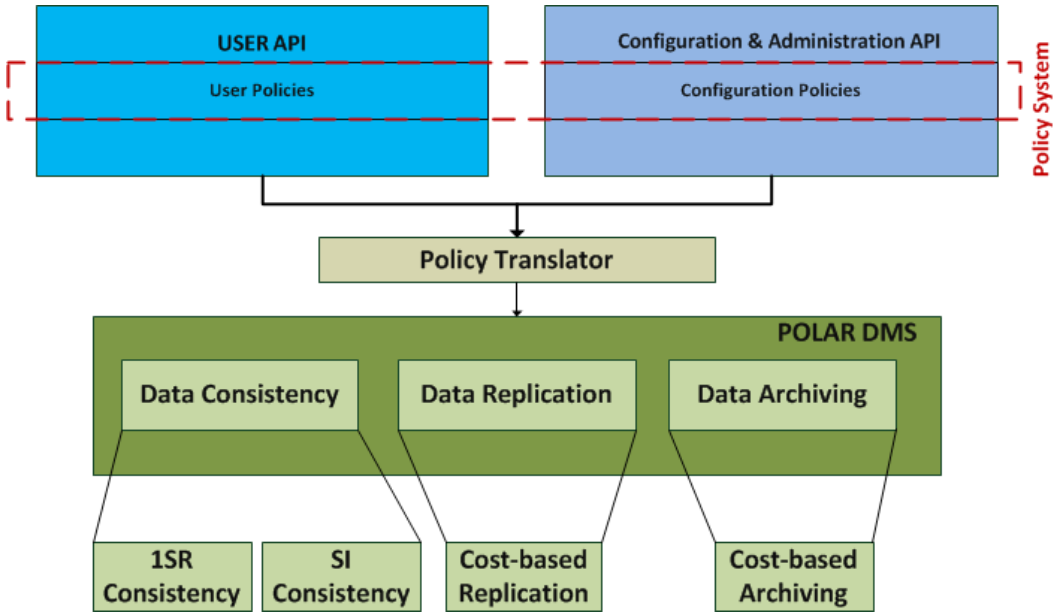
- **POLAR DMS must also handle policy conflicts!**



Modularity in POLAR DMS

Build your own DMS by composing it of modules

- **Service Provider:**
  - **Application - Developers:** Specify policies at transaction level. The system may need to adjust at runtime (load modules, etc.)
  - **Application - Architects:** Choose the modules you need for your architecture or specify application-wide policies
- **Cloud DMS Provider**
  - **Administrators:** Specify system policies and configure the DMS



Cost-effective & Policy-based Data Management

- POLAR DMS provides
  - Policy Mechanism
  - Modular Architecture

- In the context of POLAR DMS
  - Analyse a concrete subset of possible data management policies
  - Data management aspects: **consistency, replication & archiving**

Agenda

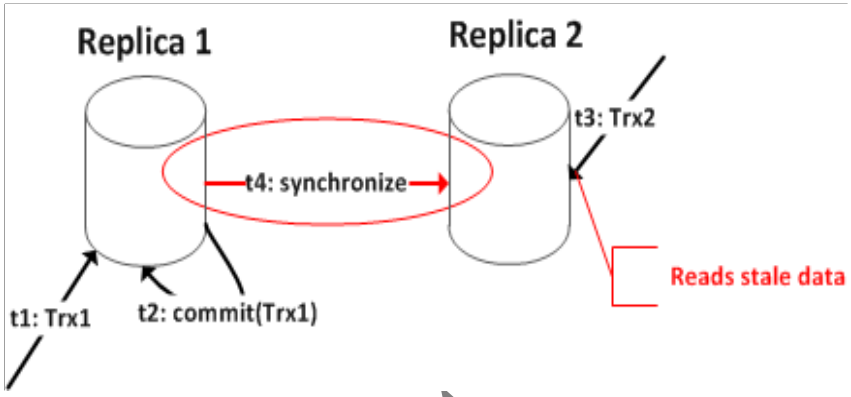
- Policy-based and modular Data Management Systems (POLAR DMS)
- Cost-effective & Policy-based Data Management
  - Data Consistency
  - Data Replication
  - Data Archiving
- Architecture of POLAR DMS
- Conclusion

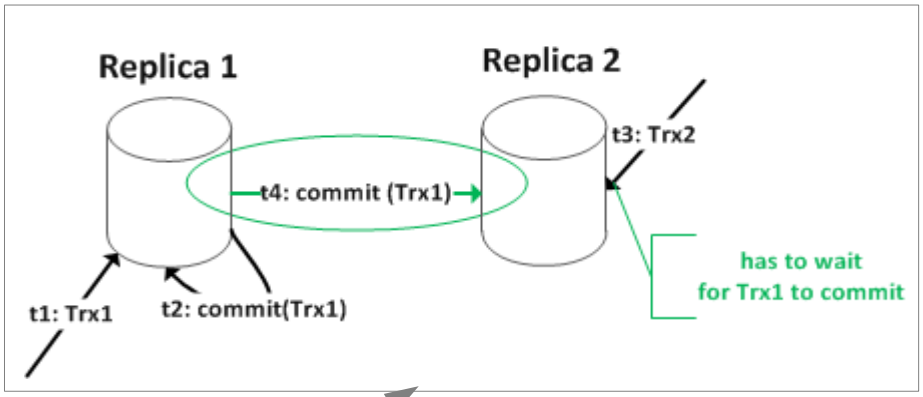
Agenda

- Policy-based and modular Data Management Systems (POLAR DMS)
- Cost-effective & Policy-based Data Management
  - Data Consistency
  - Data Replication
  - Data Archiving
- Architecture of POLAR DMS
- Conclusion

Data Consistency

- Data Consistency Level
  - A contract between DMS provider and customer
  - Customer: From application developer to end-customer
- A range of consistency levels exists
  - Strong consistency: Low availability, but more “user-friendly”
  - Weak Consistency: High availability, but less “user-friendly”





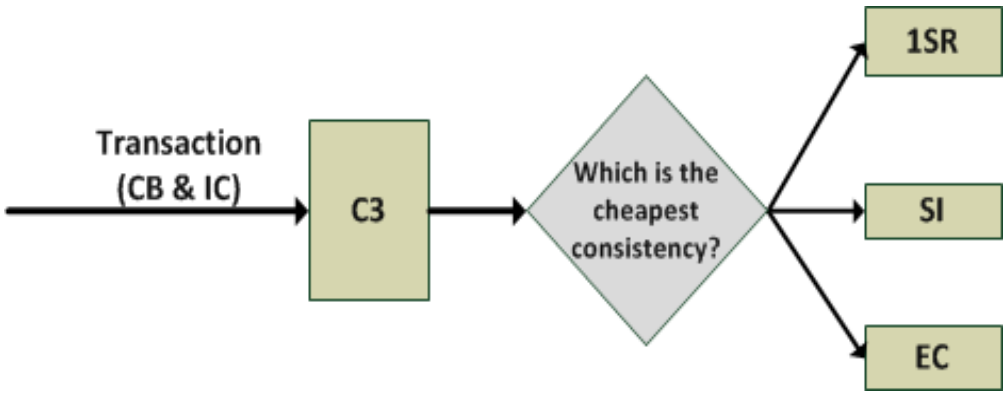
Data consistency in the Cloud

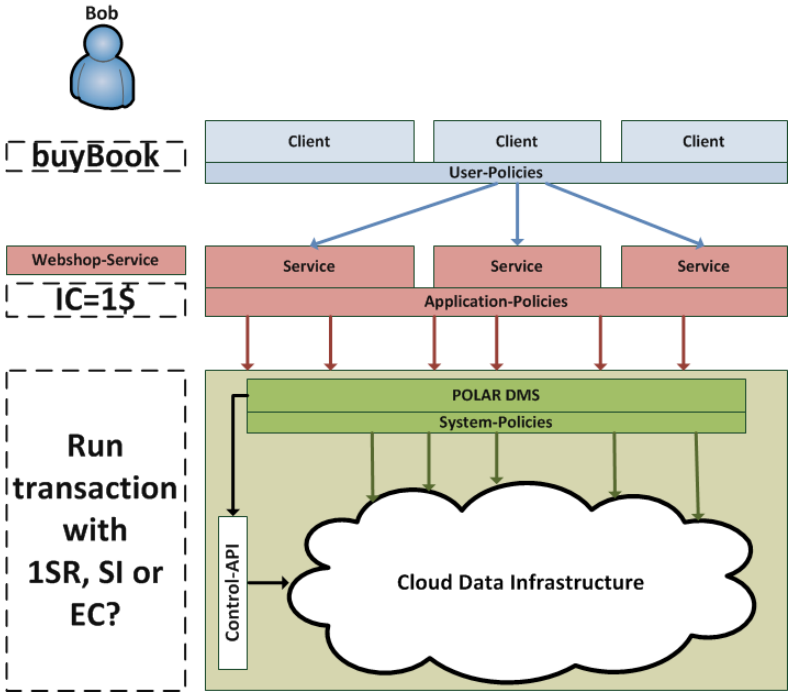
- Current Cloud Data Systems provide weak consistency
  - More oriented towards scalability
- If strong consistency needed, use traditional DBMS
  - Are less scalable due to the “consistency overhead”
- Disadvantages of current solutions
  - Different DBMS types for different consistency levels
  - No way to further influence behaviour of data consistency via policies!
- Kraska et al [3]: Consistency rationing
  - Categorize data based on their consistency requirements: 1SR, EC, Adaptive

Our approach: **Adaptive consistency based on policies at transaction level!**

Cost-based Consistency Control - C3

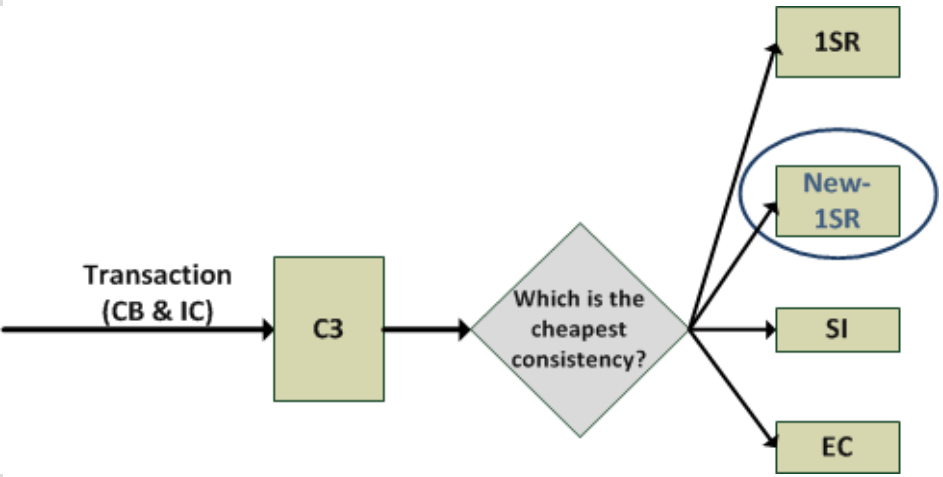
- Idea:
  - Implement a range of consistency levels
  - Provide a policy API at transaction level
  - **Adjust consistency at runtime based on the policies - C3 [1]**
- Current Consistency Levels (CL): 1SR, SI and EC
  - Cost models for 1SR, SI and EC defined in [2]
  - 1SR & SI generate no or rarely inconsistencies, but **are expensive**
  - EC is cheap, but may **generate high inconsistency costs**
- Policies: **Cost-budget (CB)** for a transaction & **inconsistency costs (IC)**

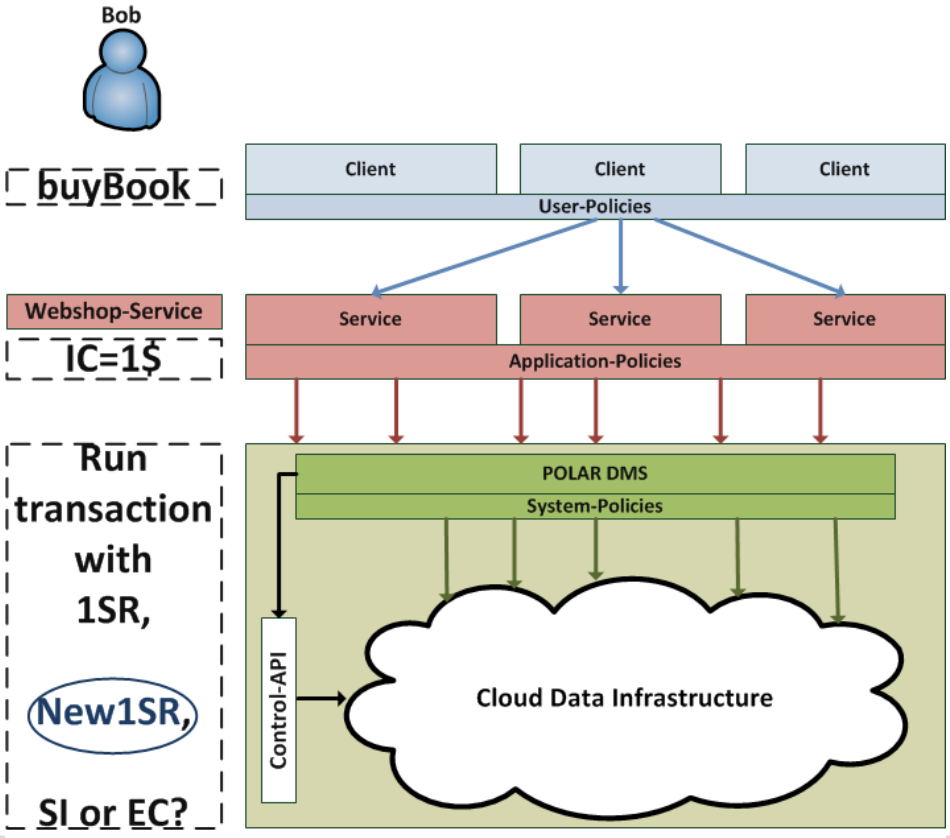




C3 Architecture

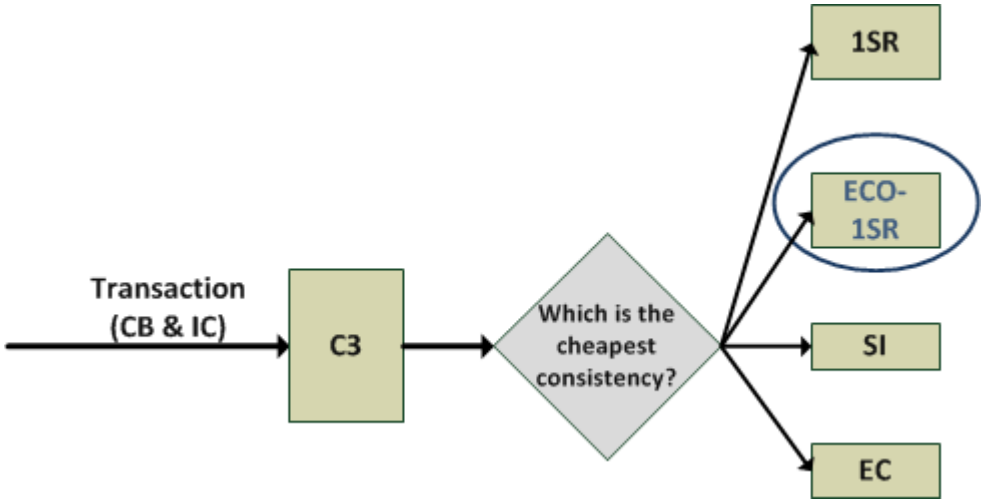
- Provides an API at transaction level (CB, IC)
  - Different combinations possible
  - Enforce a consistency level by setting $IC = \infty$
- Is based on a modular architecture
  - Implement new protocols
  - Define their cost models
  - C3 will choose the “cheapest” consistency protocol





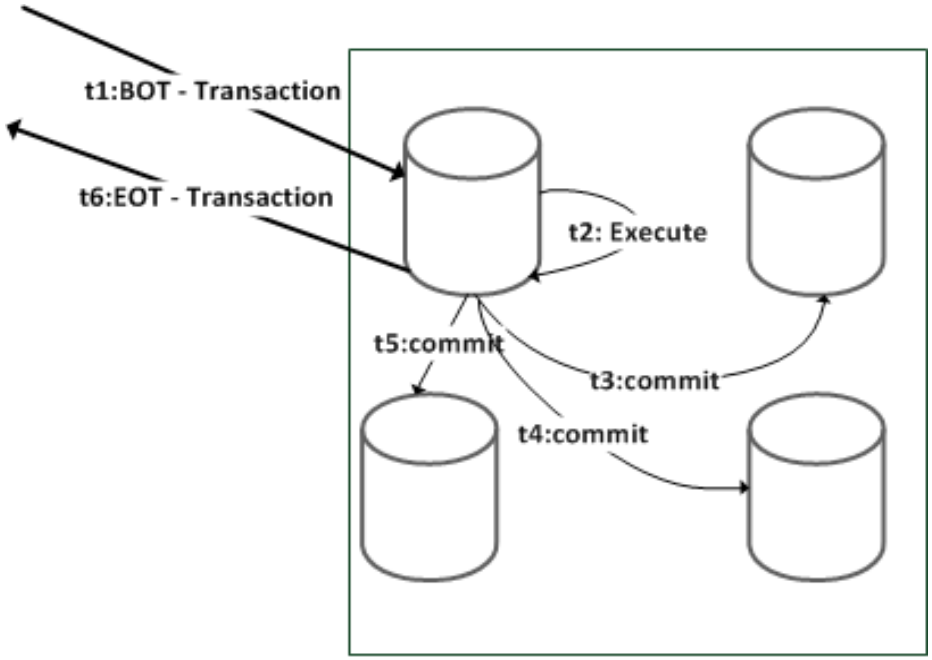
ECO-1SR

- C3: Provides a “meta-consistency” level
  - improve transaction costs & performance by switching between concrete consistency levels
- Can the concrete levels be improved?
  - Again, **based on policies**
  - Work-in-Progress: **E**fficient & **c**ost-**o**ptimized **1SR** → ECO-1SR



ECO-1SR vs. Traditional 1SR

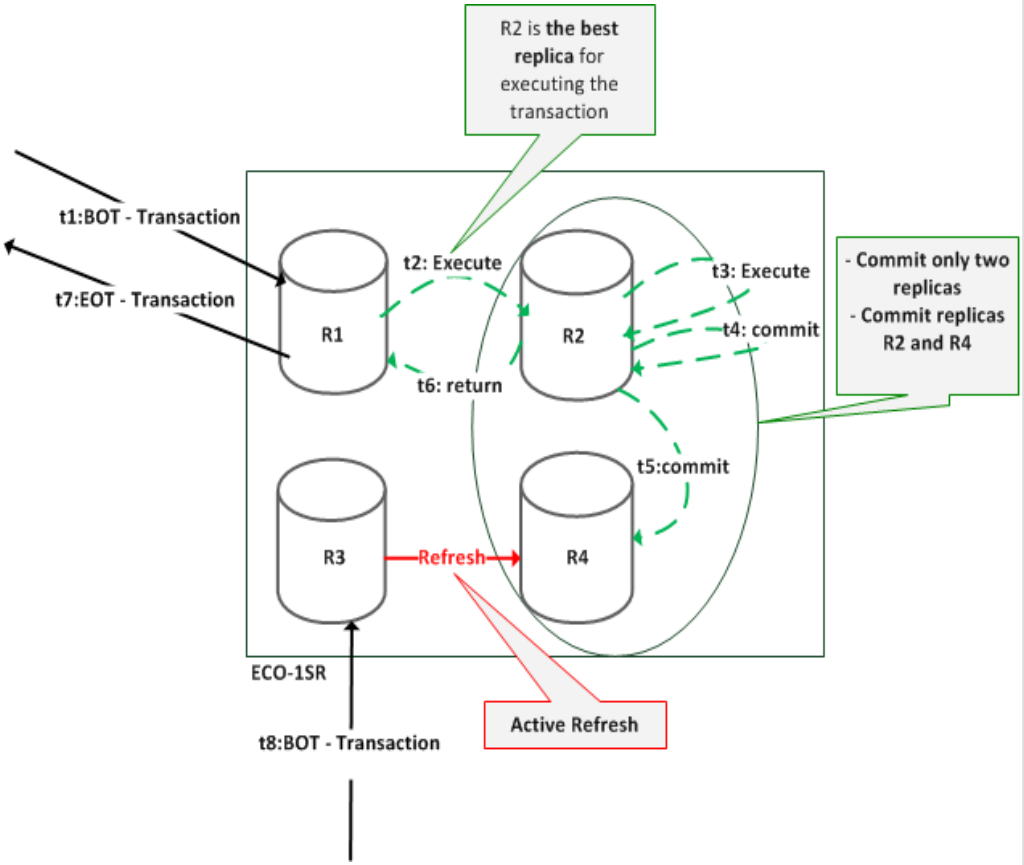
- Traditional implementation of 1SR
  - Two-Phase-Locking (2PL) and Two-Phase-Commit (2PC)
  - All replicas synchronized eagerly before transaction returns to the user
- ECO-1SR
  - **Optimize transaction execution**
  - **Optimize transaction commit:** combination of eager & lazy replica synchronization



Traditional 1SR

ECO-1SR vs. Traditional 1SR

- Traditional implementation of 1SR
  - Two-Phase-Locking (2PL) and Two-Phase-Commit (2PC)
  - All replicas synchronized eagerly before transaction returns to the user
- ECO-1SR
  - **Optimize transaction execution**
  - **Optimize transaction commit:** combination of eager & lazy replica synchronization



ECO-1SR Optimizations

- Transaction execution: **which** replica is best suited for the transaction execution
  - Choose the replica with highest capacity and freshness, and lowest cost
- Transaction commit: **how many** and **which replicas** to synchronize eagerly
  - **How many:** Depending on data popularity (i.e. transaction popularity)
  - **Which:** Choose the one with highest capacity, highest popularity and lowest cost
  - All other replicas updated lazily
  - Active refresh in the case transactions access stale data

→ The 'right' choice reduces the overhead for providing 1SR

Agenda

- Policy-based and modular Data Management Systems (POLAR DMS)
- Cost-effective & Policy-based Data Management
  - Data Consistency
  - Data Replication
  - Data Archiving
- Architecture of POLAR DMS
- Conclusion

Cost-based Data Replication

- How to provide **efficient** and **cost-optimized replication**?
  - What is the **best number** of replicas ?
  - What is the **best replica type** ?
  - What is the **best replica location** ?
- **Example 1**
  - In the next period many high important transactions expected
  - High importance: **high profit or high penalty if constraints violated**!
  - Should be more replicas be generated? What replica types are more cost-effective?
- **Example 2**
  - Legal issues: Data can be stored/replicated only in specific locations
  - Specify replication policies per data-object or application

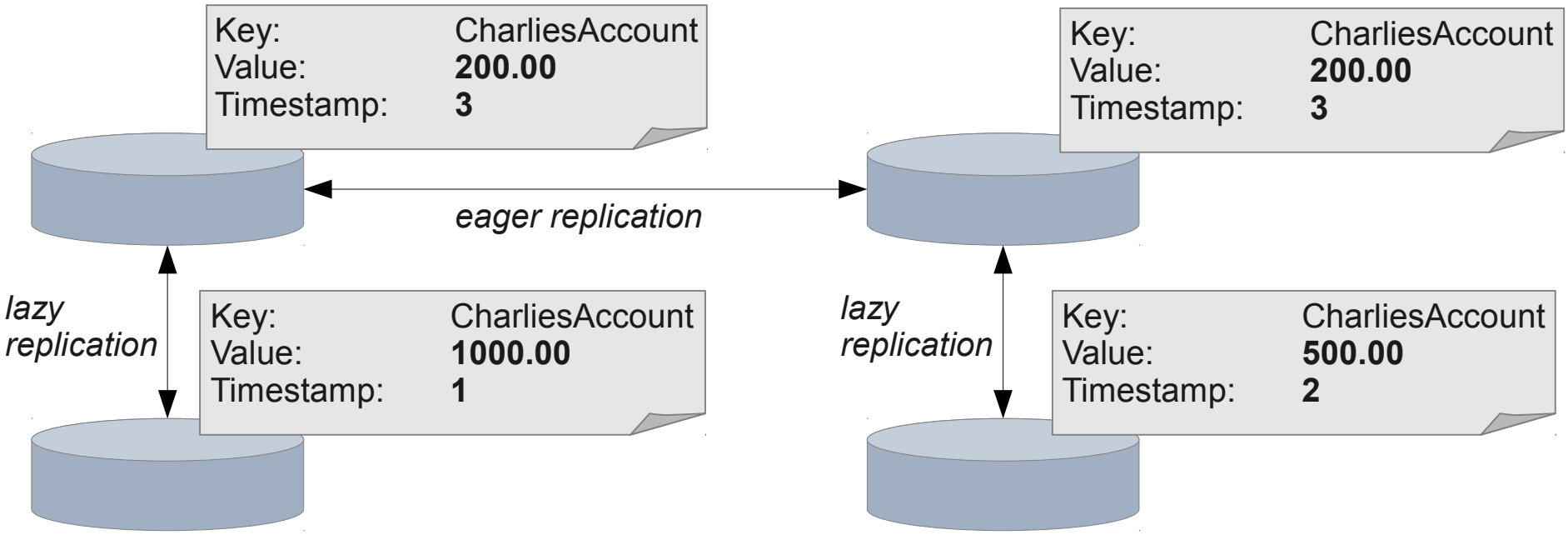
Agenda

- Policy-based and modular Data Management Systems (POLAR DMS)
- Cost-effective & Policy-based Data Management
  - Data Consistency
  - Data Replication
  - Data Archiving
- Architecture of POLAR DMS
- Conclusion

Data Archiving - Motivation

- Assumption: freshness-aware key-value store
  - → timestamped/versioned data items
  - → freshness-centric operations

- Observation:
  - concurrently different versions of the same data item by design
  - different applications/clients: different freshness requirements



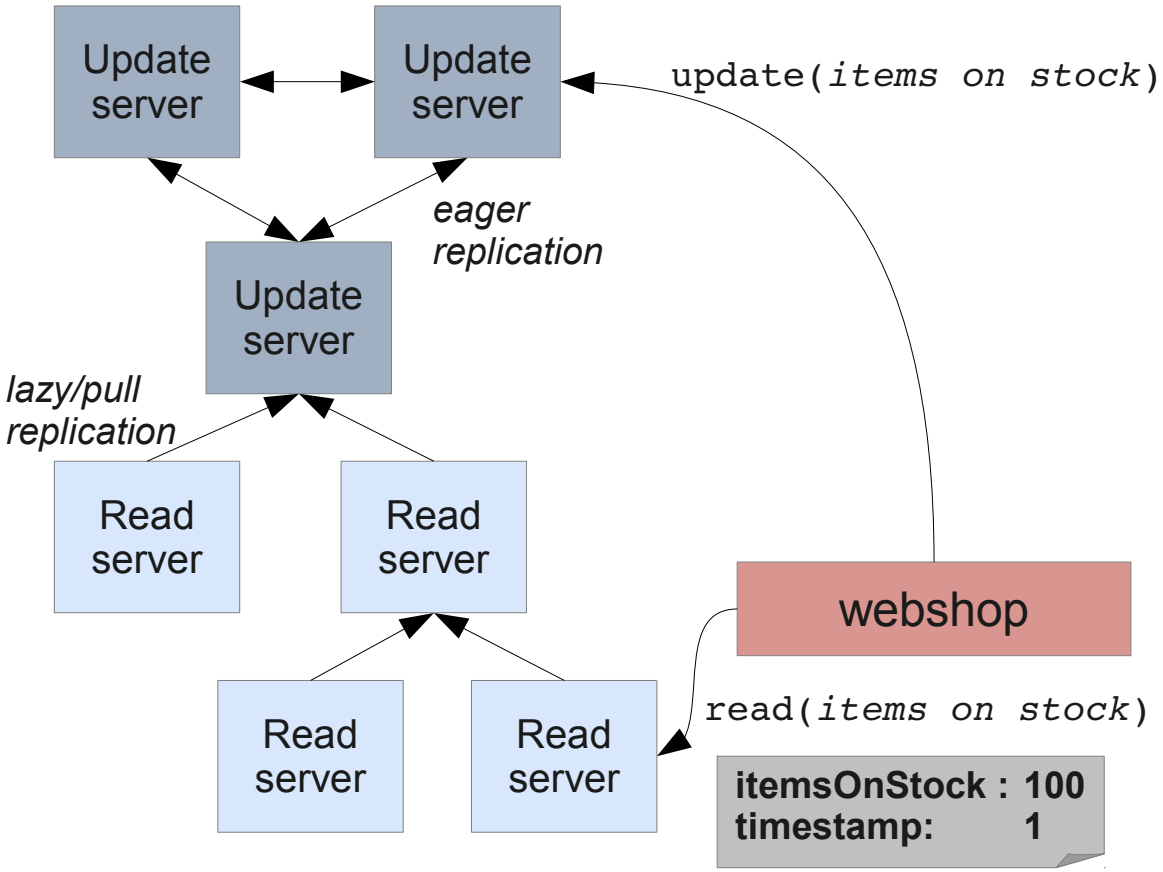
Data Archiving - Motivation

- Assumption: freshness-aware key-value store
  - → timestamped data items
  - → freshness-centric operations
- Observation:
  - concurrently different versions of the same data item by design
  - different applications/clients: different freshness requirements
- Idea: not deal with, but exploit these properties
- Goals:
  - archive cloud data in situ
  - allow for
    - policy enforcement
    - more freshness-centric operations
    - enhanced replica placement

K/V Store Infrastructure

- Example:
  freshness-aware K/V store for webshop
- Few writes, many reads

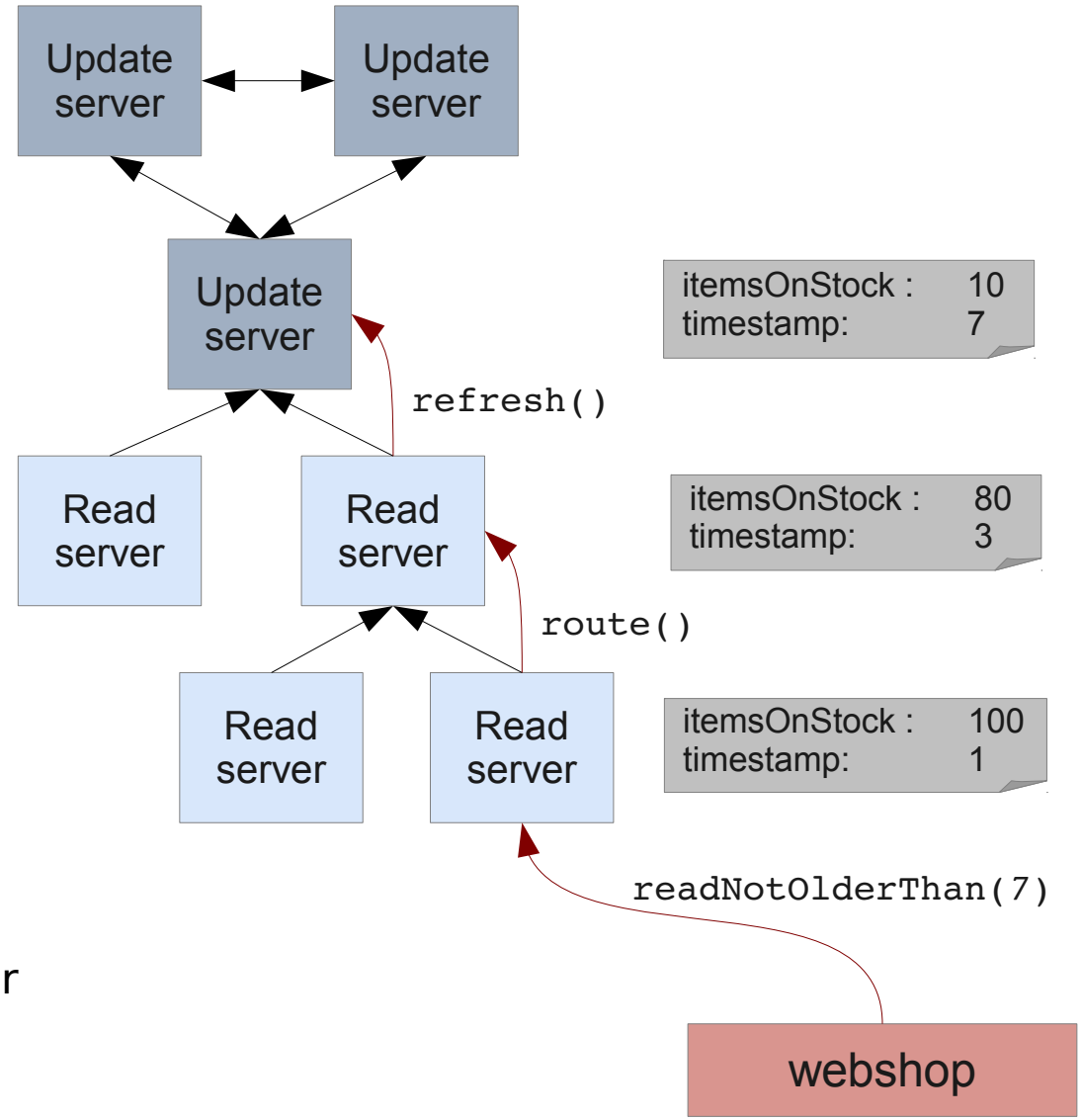
  → few update, many read servers
- Read-semantics
  - `read()`
    - latest **local** data sufficient



K/V Store Infrastructure II

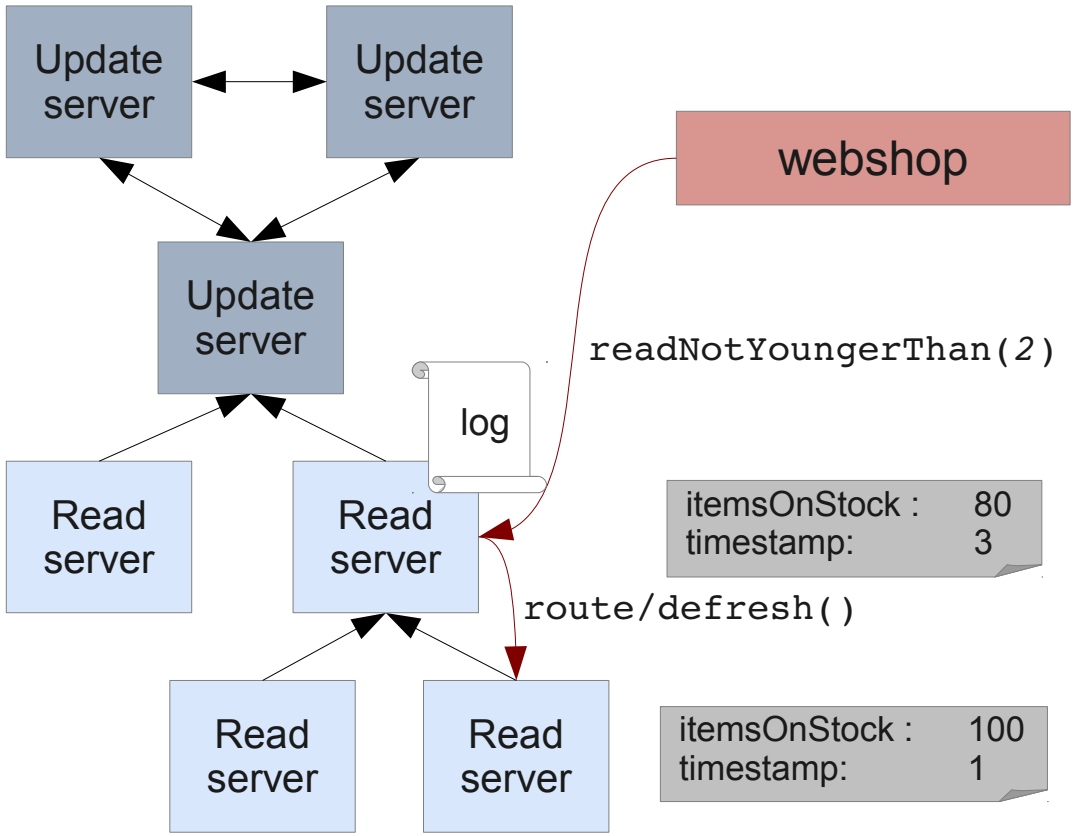
- Example:
  freshness-aware K/V store for a webshop
- Few writes, many reads

  → few update, many read servers
- Read-semantics
  - `read()`
    - latest **local** data sufficient
  - `readNotOlderThan(`*`time t`*`)`
    - when certain **freshness level** required
    - possibly `route` / `refresh` necessary
    - operation is feasible, however potentially expensive



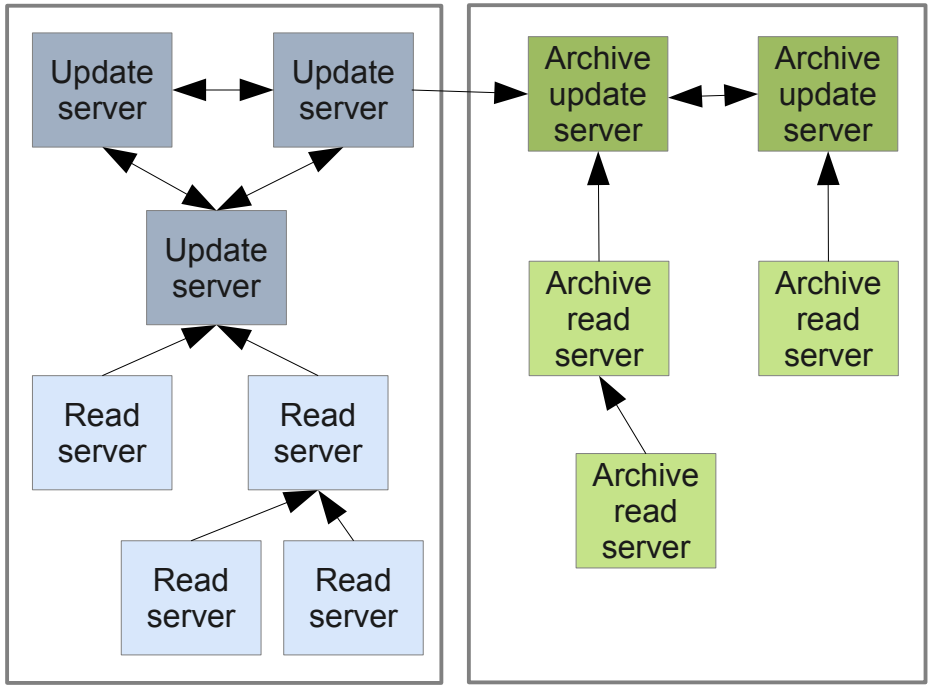
K/V Store Infrastructure III

- Other read-semantics are necessary as well.
- However, they are either
  - **expensive** or
  - **impossible**. *(Or even both!)*
- Examples:
  - `readNotYoungerThan(t)`
    - "How is the stock of a book developing?"
  - `readBetween(t1, t2)`
    - "How was it yesterday?"
  - `readAllBetween(t1, t2)`
    - "How exactly did it fluctuate during our sales week?"
- Data has to be reconstructed via
  - **local logs** or
  - **route**/"**defresh**".

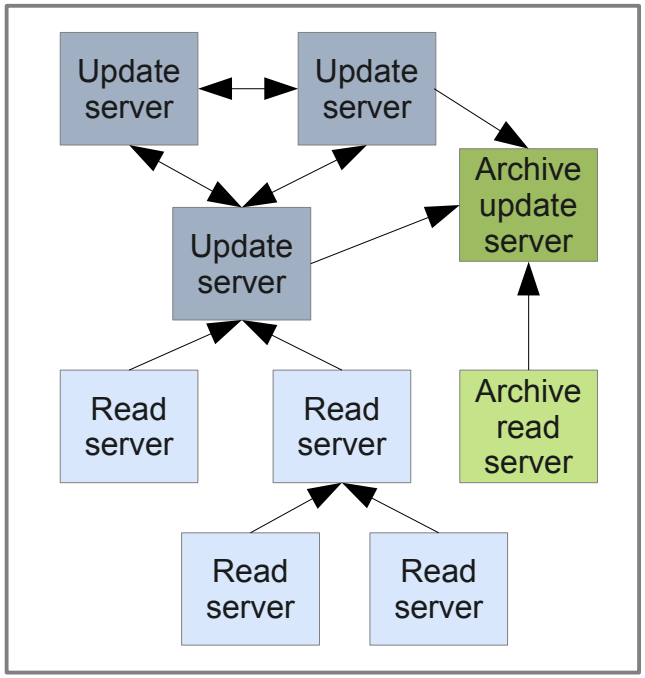


Archive Infrastructure

- An archive helps “rolling back time”
- Different possible archive layouts possible:



Dedicated archive



Integrated archive

Data Archiving with Policies

- Control over archiving is necessary
  - Which data?
  - How fast?
  - How widely spread?
  - What about successive updates?
  - How long? → minimal, maximal TTL
  - Who may access data?
  - How much $ to spend?
  - etc.

→ Policies are needed

Data Archiving with Policies

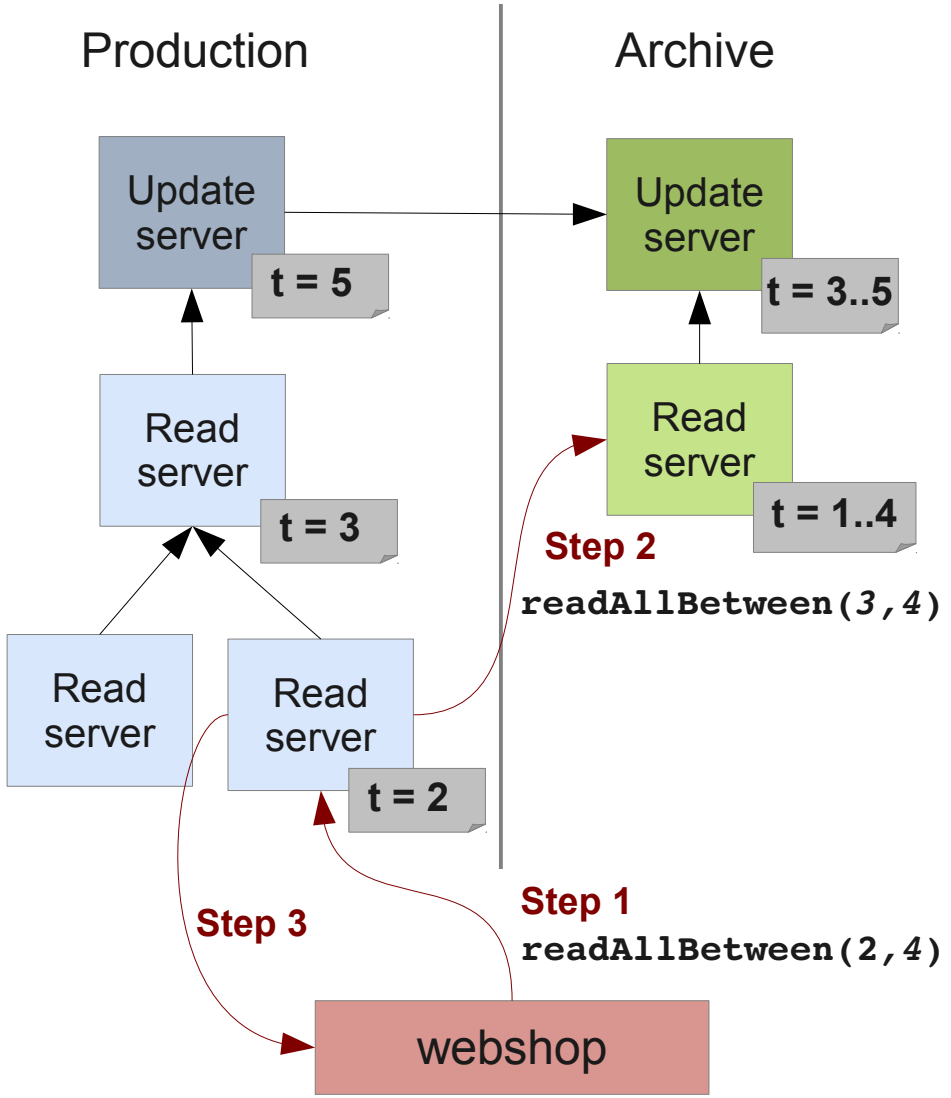
- Control over archiving is necessary
  - Which data?
  - How fast?
  - How widely spread?
  - What about successive updates?
  - How long? → minimal, maximal TTL
  - Who may access data?
  - How much $ to spend?
  - etc.

Data Policy

Key:	**CharliesAccount**
Speed:	**eager**
Spread:	**max**
Successive updates:	**infinite**
TTLmin:	**10 years**
TTLmax:	**10 years**
Access:	**charlie;accountManager**
Budget:	**infinite**

Read*-Semantics

- Rich(er) read*-semantics can be handled efficiently with the archive
  - without archive:
    - `read()`
    - `readNotOlderThan(t)`
  - with archive:
    - `readNotYoungerThan(t)`
    - `readBetween(t1, t2)`
    - `readAllBetween(t1, t2)`



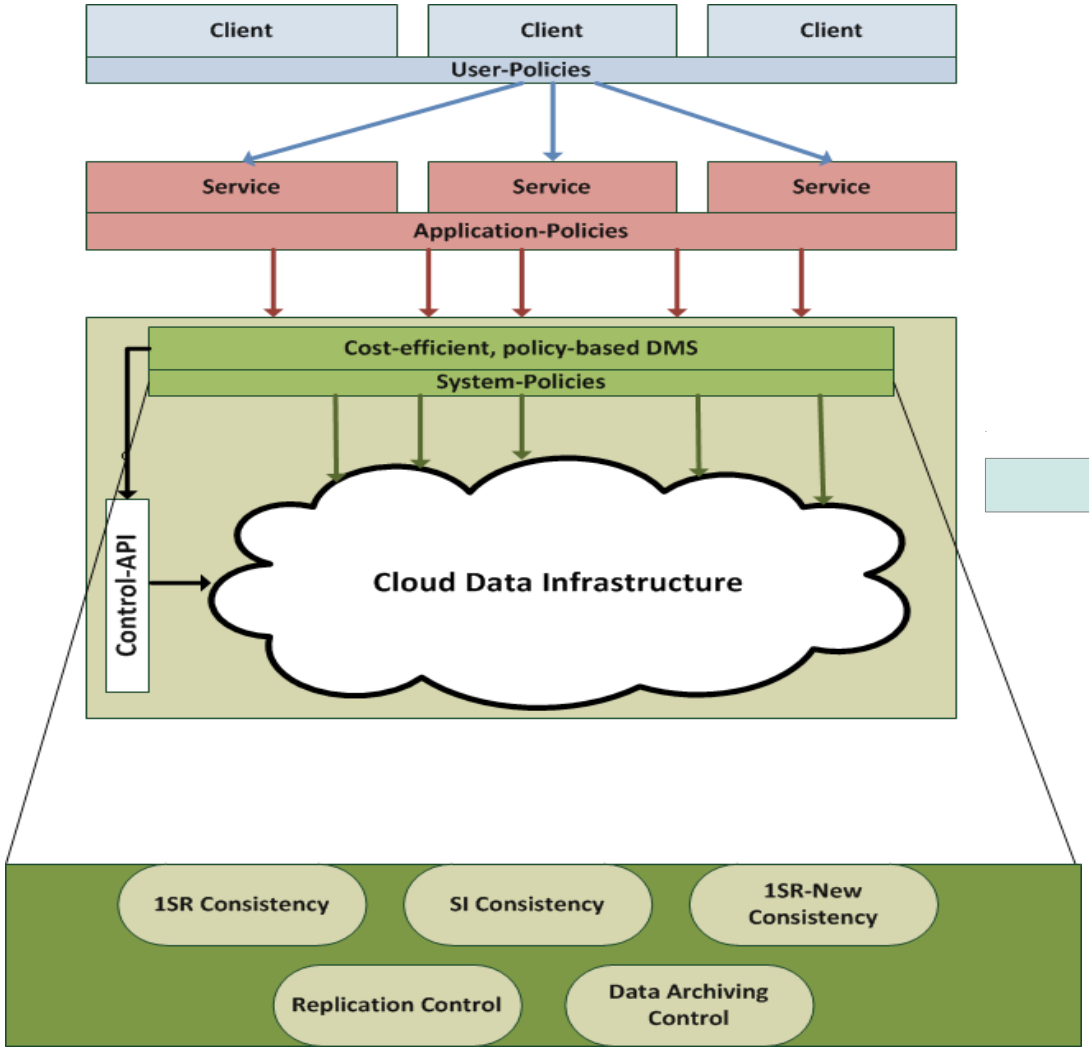
Putting the Pieces Together

- High-level, conceptual vision of modular & policy-based DMS
- Concepts of
  - Data Consistency
  - Data Replication
  - Data Archiving
- Recall: we want to reach
  - policy-based and
  - modular DMSs
- In order to put the pieces together: **we built a concrete architecture**

Agenda

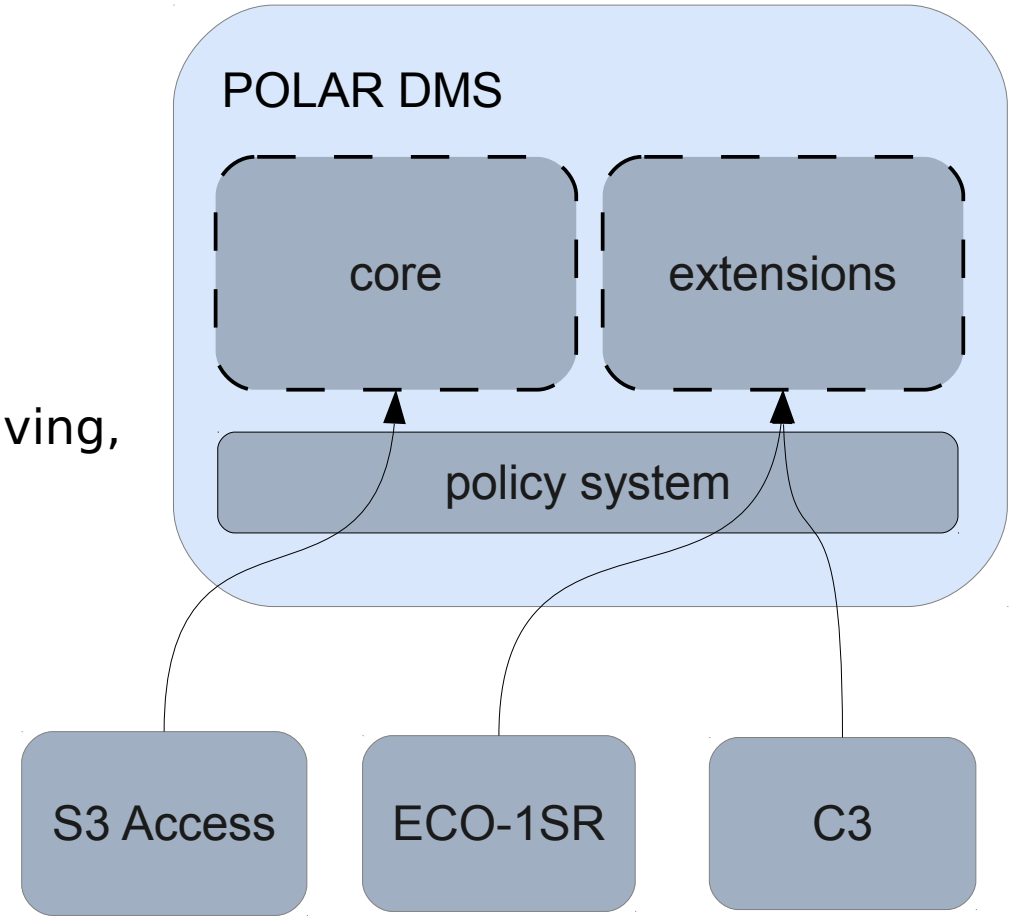
- Policy-based and modular Database Management Systems (POLAR DMS)
- Cost-effective & Policy-based Database Management
  - Data Consistency
  - Data Replication
  - Data Archiving
- Architecture of POLAR DMS
- Conclusion

Architecture of POLAR DMS



Architecture of POLAR DMS

- Based on OSGi
  - Module-based framework for Java
  - Facilitates run-time changes of modules
- Basic architecture
  - Implement **core** modules: Data Access, Routing, etc.
  - Implement **additional** functional modules: C3, ECO-1SR, Data Archiving, etc.
- Policy controlled module selection
  - Finding optimal module selection and configuration

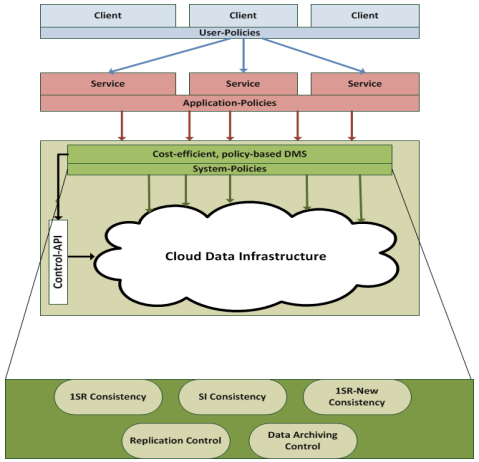


Agenda

- Policy-based and modular Data Management Systems (POLAR DMS)
- Cost-effective & Policy-based Data Management
  - Data Consistency
  - Data Replication
  - Data Archiving
- Architecture of POLAR DMS
- Conclusion

Conclusion

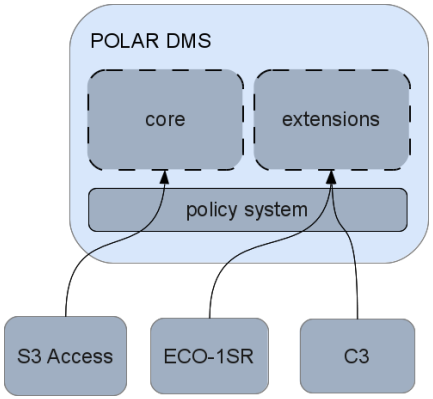
- POLAR DMS consists of a unified model for:
  - **categorizing**,
  - **assessing** and
  - **creating** Data Management Systems (DMS)



- This is achieved by a policy-driven framework architecture

- We built concrete functional modules, enabling cost-effective and policy-based
  - Data Consistency
  - Data Replication
  - Data Archiving

- We provide a concrete architecture of POLAR DMS



Thank You!

Questions?

References

- [1]: Ilir Fetai and Heiko Schuldt, *“Cost-Based Data Consistency in a Data-as-a-Service Cloud Environment”.* Proceedings of the 5th International Conference on Cloud Computing (CLOUD 2012), Honolulu, HI, USA, 2012/6.

- [2]: Ilir Fetai and Heiko Schuldt, *“Cost-Based Adaptive Concurrency Control in the Cloud”.* Technical Report, Department of Mathematics and Computer Science, University of Basel, 2012/2.

- [3]: Kraska et al, “Consistency rationing in the cloud: pay only when it matters”. Proc. VLDB Endow., 2009.