Scalable and Secure Concurrent Evaluation of History-based Access Control Policies

Maarten Decat, Bert Lagaisse, Wouter Joosen
KU Leuven, Belgium
maarten.decat@cs.kuleuven.be
Access control: enforce access rules

Subjects → Read → Resources

Policies

Application
Load balancer

Application

Database

Policy Decision Point

Application

Database

Policy Decision Point

Application

Database

Policy Decision Point

Application

Database

Policy Decision Point

Application

Database

Policy Decision Point
Problem
Problem

Not all policies are read-only: history-based policies

“a user cannot watch more than 10 movies per month”

“an article cannot be shared more than 5 times”

“if a user has had access to documents of Bank A, he or she is not allowed to access documents of Bank B”

(Dynamic SoD)
Problem

Subject1 -> doc of Bank A

read subj.history  // (= [])
... // evaluate policy = permit
append “Bank A” to subj.history
return permit

Subject1 -> doc of Bank B

read subj.history  // (= [“Bank A”])
... // evaluate policy = deny
return deny
Problem

Subject1 -> doc of Bank A

read subj.history // (= [])
...
// evaluate policy = permit
append “Bank A” to subj.history
return permit

Subject1 -> doc of Bank B

read subj.history // (= [])
...
// evaluate policy = permit
append “Bank B” to subj.history
return permit

Read-write conflict
Goal

- We need concurrency control

- Possible approach: model policy evaluations as transactions on the underlying database

- However: serial equivalence does not scale
  - E.g., MySQL: single master server for transactions
  - E.g., Cassandra, MongoDB: compare-and-set on single database elements

⇒ we need a domain-specific form of concurrency control to achieve both serial equivalence and scalability
Approach
Approach

When resource.owner == “Bank B”,
apply DenyOverrides to

Deny if
“Bank A” in subject.history

Permit performing
append(“Bank B”, subject.history)

Goal: detect and contain read-write conflicts for
the same attribute in parallel policy evaluations
Possible tactics

1. Locks
2. (Multi-version) time-stamp ordering
3. Optimistic concurrency control
   - Check for conflicts before committing
   - Roll back any updates in case of conflict
   - Reasons:
     - Fits the structure of a policy evaluation: all attribute updates are performed after all reads
     - Does not block policy evaluations
     - Does not rely on database functionality
Basic approach

Optimistic concurrency control

Policy decision point

Application

Coordinator

Worker

Attribute db

1. Application
2. Coordinator
3. Worker
4. Attribute db
5. Coordinator
6. Worker
7. Coordinator
8. Coordinator
9. Application
Centralized coordinator
Scalable distributed coordinator

- Observations:
  - Every policy evaluation reasons about exactly 1 subject and 1 resource

⇒ Distribute these two parts over two collaborating coordinators
Responsible coordinator based on hash of id of subject/resource

Scalable concurrency control
Scalable concurrency control
Scalable concurrency control
Scalable concurrency control
Performance evaluation
Prototype and test set-up

- Prototype:
  - STAPL policy language [https://github.com/stapl-dsl/]
  - Akka actor framework for concurrency and distributed communication

- Test set-up:
  - Run tests on VMs until statistically relevant
  - Employ realistic policy from e-health case study [https://people.cs.kuleuven.be/~maarten.decat/acsac2015/]
Results

[Graph showing throughput in requests per second against the number of parallel clients.]

- Client
- Coordinator
- Worker
Results

![Diagram showing client, coordinator, and worker nodes with throughput graphs for 2 CPUs, 4 CPUs, and 8 CPUs.](image-url)
Results

![Diagram showing the relationship between the number of distributed coordinators and throughput. The diagram includes a hierarchical structure with clients and coordinators, with workers at the bottom level. The graph on the right side plots throughput (requests per second) against the number of distributed coordinators. The throughput increases linearly as the number of coordinators increases.]
Results

Asymptotic maximal latency
Conclusions
Conclusions

- **Goal:** applying policy-based access control on large-scale distributed systems
- **Focus:** address concurrency issues that can arise for history-based access control policies
- **Contribution:** a domain-specific scheme for concurrency control that:
  - prevents incorrect decisions
  - can *scale* to a large number of machines
  - introduces a limited and asymptotically *bounded* latency overhead
Scalable and Secure Concurrent Evaluation of History-based Access Control Policies

Maarten Decat, Bert Lagaisse, Wouter Joosen
KU Leuven, Belgium
maarten.decat@cs.kuleuven.be