Cassandra: Distributed Access Control Policies with Tunable Expressiveness

Moritz Y. Becker, and Peter Sewell, 5th IEEE POLICY, 2004

Yue Zhang

yzhang@sis.pitt.edu

September 16, 2009

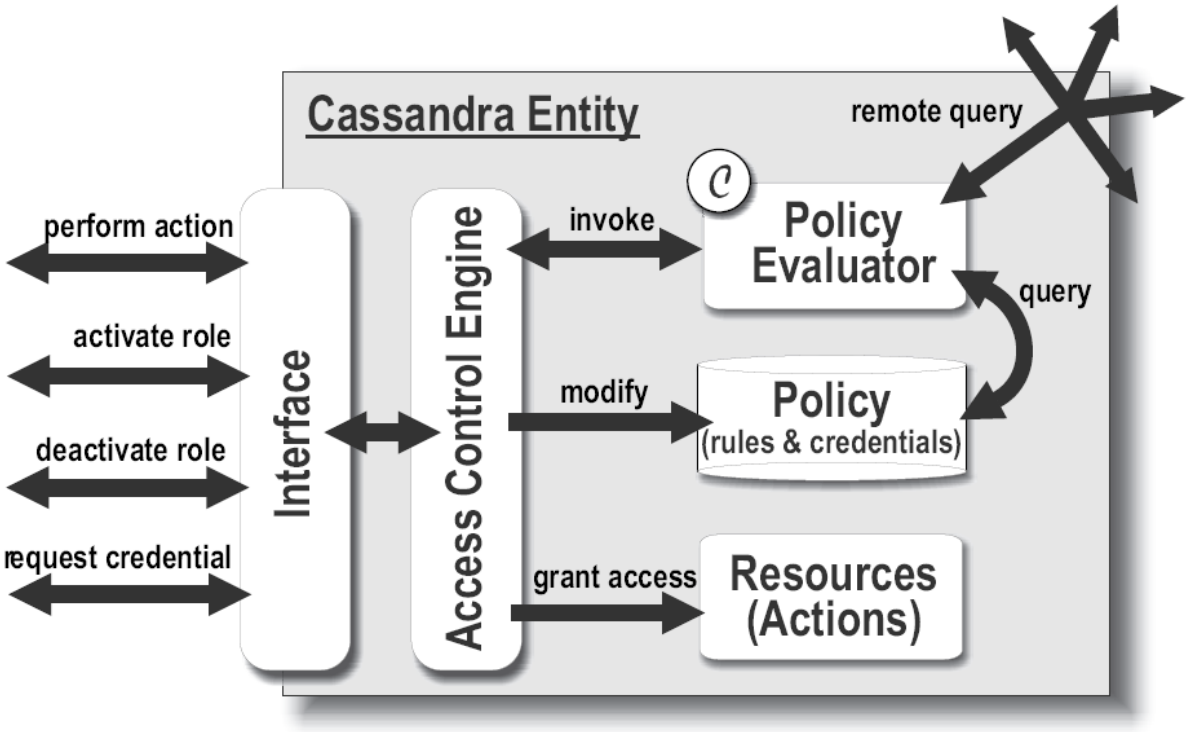
What's wrong with RT?

- Extensions are ad-hoc
  - further changes?
  - implementation needs to be changed
- Less support for constraints
  - fixed syntax and semantics
  - very limited support for constraints, e.g. ranges on parameters of roles
  - constraints between parameters are not supported

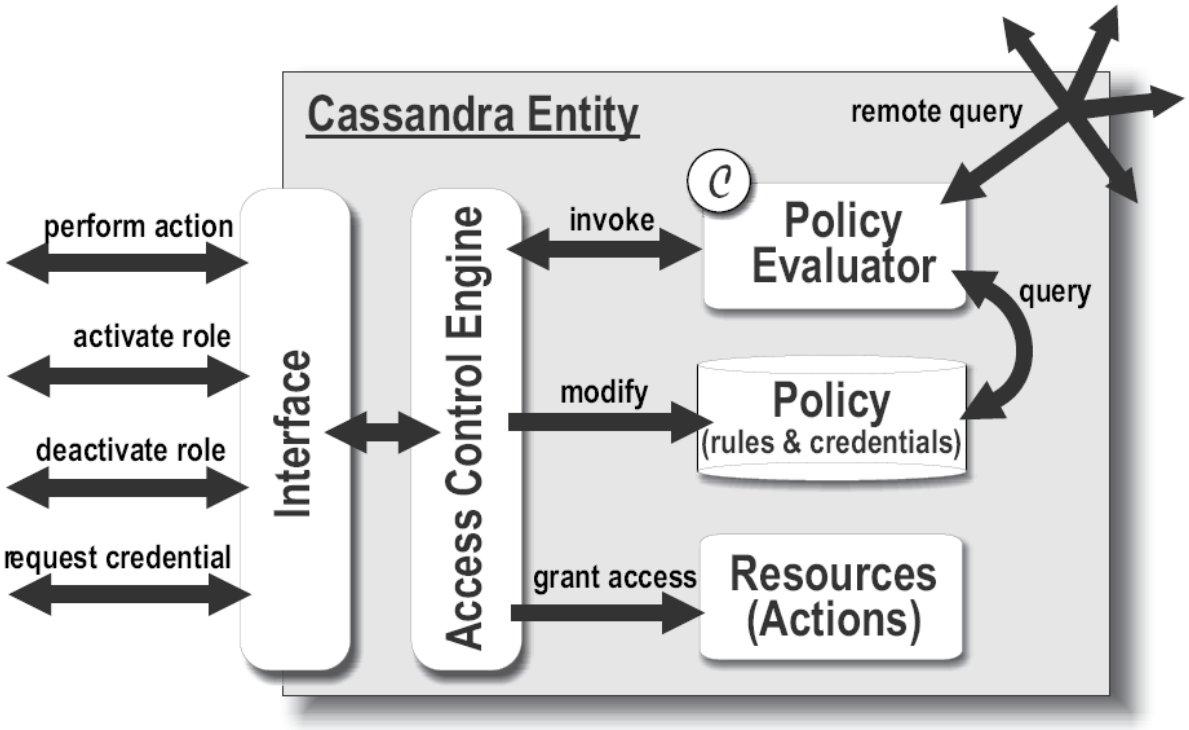
Cassandra: access control policies with tunable expressiveness

- Features:
- Tunable Expressiveness
  - 5 basic predicates
  - relies on Datalog_C
  - Tune the expressiveness through different constraint domains
  - double edged sword!!!
- A complete system
  - not only a policy specification language
  - system architecture
  - protocol for credential discovery and trust negotiation

Svstem Architecture



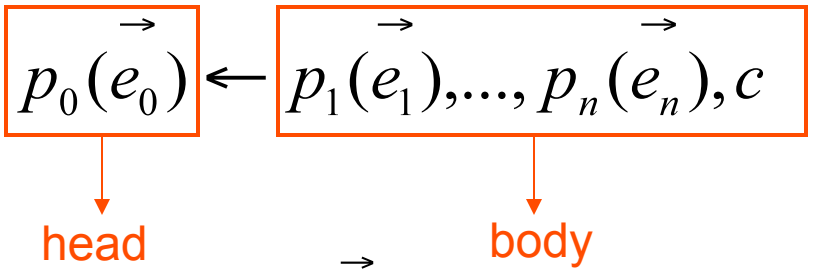
System Architecture



Discussion:
How does the user know that she needs to request to activate some role before she can request to perform some action?

Rules in Cassandra

- Expressed by Catalog_C, C is a constraint domain
- Expression:



- p_i is the predicate name, $\vec{e_i}$ is the values matching the parameters of the predicate
- *c* is a constraint on the parameters occurring in the rest of the rule, and *c* is from *C*.
- If the body of a rule is empty, this rule is actually a credential rule to represent a credential.

Basic Predicates

- permits (*E*, *A*)
  - specifies *E* is permitted to take action *A*
- canActivate (*E*, *R*)
  - specifies *E* can activate role *R* (*i.e. E is a member of R*)
- hasActivated (*E*, *R*)
  - specifies entity *E* is currently active in role *R*
- canDeactivate (*E*, *V*, *R*)
  - specifies entity *E* can revoke *V*'s role *R*
- isDeactivated (*E*, *R*)
  - specifies automatically triggered role revocation
- canReqCred (*E*, $I.p(\vec{x}) \leftarrow c$)
  - specifies the conditions to be satisfied before the service is willing to issue and disclose a credential $I.p(\vec{x}) \leftarrow c$ to entity *E.*

Decentralized Attributes

- Each predicate is of the form $loc@iss.p(\vec{e})$
  - *loc* is the location storing the predicate
  - *iss* is the issuer of the predicate
- If a predicate appears in the body of a rule in *E*'s policy:
  - *loc* is equal to *E*: it is deduced locally from *E*'s policy
  - *loc* is not equal to *E*: the authority should be queried from a foreign entity *loc*, so *E* requests a credential $iss.p(\vec{e})$ from *loc*
- If a predicate appears in the head of a rule in *E*'s policy:
  - *loc* and *iss* are called location and issuer of the rule, and are always identical except when the rule is a credential (which means *loc* holds a foreign credential signed by a different entity *iss*)

Access Control Semantics

- Performing an action
  - *E* is granted to perform action *A* on *S*'s Cassandra service if:
  - permits (*E*, *A*) is deducible from *S*'s policy
- Role activation
  - *E* is granted to activate role *R* on *S*'s Cassandra service if:
  - canActivate (*E*, *R*) is deducible from *S*'s policy
- Role deactivation
  - *E* is granted to deactivate *V*'s role *R* on *S*'s Cassandra service if:
  - *V* is active in *R*, and canDeactivate (*E*, *V*, *R*) is deducible from *S*'s policy
  - may trigger more deactivations
- Requesting credentials
  - *E* requests the credential $I.p(\vec{x}) \leftarrow c$ from *S*
  - *S* computes the answer to the query canReqCred (*E*, $I.p(\vec{x})) \leftarrow c$. the answer is a constraint c_0 restricting the values that *x* can take
  - if *S*==*I*, compute c_1 be the answer of the query $p(\vec{x}) \leftarrow c_0$. Then if c_1 is satisfiable, $S.p(\vec{x}) \leftarrow c_1$ is sent to E
  - if *S* != *I*, *S* sends *E* all her credentials of the form $I.p(\vec{x}) \leftarrow c_2$, such that c_2 is at least as restrictive as c_0

Standard Policies

- Cassandra is claimed to be able to express a wide range of policies using its small language construct, including:
- Role validity periods
- Auxiliary roles
- Role hierarchy
- Separation of duties
- Role Delegation
- Automatic trust negotiation & credential discovery

Role validity periods

- Scenario:

  - a certified doctor (with certification issued at time *t*) is also a member of role Doc() if *t* is at most 1 year ago.

- Policy (rule):

canActivate(*x*, Doc()) ←

canActivate(*x*, CertDoc(*t*)),

CurTime() – Years(1) ≤ *t* ≤ CurTime()

An example of constraint, not supported by RT

Auxiliary roles

- Scenario

  - a logged-in user can read a file provided that the system can deduce she is the owner of that file

- Policy (rule):

  permits(*x*, Read(*file*))←

  hasActivated(*x*, Login()),

  canActivate(*x*, Owner(*file*))

Role hierarchy

- Scenario:

  members of the Engineer role are automatically also members of the Employee role in the same department, i.e., Engineer is senior to Employee

- Policy (rule):

  canActivate(x, Employee(dep))←

  canActivate(x, Engineer(dep))

Separation of duties

- Scenario:

  an Authoriser of a payment must not have activated the Init role for the same payment

- Policy (rule):

  canActivate(*x*, Authoriser(*payment*))←

  countInitiators(*n*, *x*, *payment*), *n*=0

  countInitiators(count<*z*>, *x*, *payment*)←

  hasActivated(*z*, Init(*payment*)), *z*=*x*

user-defined predicate

an aggregate rule

definition of countInitiators()

AWKWARD!!!

Role Delegation

- Scenario:

  an administrator *x* can delegate her Adm() role to somebody else by activating the DelegateAdm() role for the delegatee y. The delegatee y can then subsequently activate the administrator role

- Policy (rule):

  canActivate(x, DelegateAdm(*y*, *n*))←hasActivated(x, Adm(*z*, *n*))

  canActivate(y, Adm(*x*, *n*'))←hasActivated(*x*, DelegateAdm(*y*, *n*)), 0 ≤ *n*' < *n*

  delegation chain: z → x → y

- Scenario:

  the delegated role is automatically revoked if the delegation role of the delegator is deactivated

- Policy (rule):

  isDeactivated(*y*, Adm(*x*, *n*'))←isDeactivated(*x*, DelegateAdm(*y*, *n*))

- Scenario:

  only the delegator can deactivate a delegation role

- Policy (rule):

  canDeactivate(*x*, *z*, DelegateAdm(*y*, *n*))←*x*=*z*

- Scenario:

  every administrator whose rank is at least as high as the delegator can deactivate a delegation role

- Policy (rule):

  canDeactivate(*x*, *z*, DelegateAdm(*y*, *n*))←hasActivate(*x*, Adm(*w*, *n*')), *n* ≤ *n*'

Role Delegation

- Scenario:

  an administrator *x* can delegate her Adm() role to somebody else by activating the DelegateAdm() role for the delegatee y. The delegatee y can then subsequently activate the administrator role

- Policy (rule):

  canActivate(x, DelegateAdm(*y*, *n*))←hasActivated(x, Adm(*z*, *n*))

  canActivate(y, Adm(*x*, *n*'))←hasActivated(*x*, DelegateAdm(*y*, *n*)), 0 ≤ *n*' < *n*

  delegation chain: z → x → y

- Scenario:

  the delegated role is automatically revoked if the delegation role of the delegator is deactivated

- Policy (rule):

  isDeactivated(*y*, Adm(*x*, *n*'))←isDeactivated(*x*, DelegateAdm(*y*, *n*))

- Scenario:

  only the delegator can deactivate a delegation role

- Policy (rule):

  canDeactivate(*x*, *z*, DelegateAdm(*y*, *n*))←*x*=*z*

Discussion:
Who creates the DelegateAdm() role?
Every role needs to have a corresponding delegation role in order for it to be delegated?

- Scenario:

  every administrator whose rank is at least as high as the delegator can deactivate a delegation role

- Policy (rule):

  canDeactivate(*x*, *z*, DelegateAdm(*y*, *n*))←hasActivate(*x*, Adm(*w*, *n*')), *n* ≤ *n*'

Automatic trust negotiation & credential discovery

- Scenario:

  To activate the doctor role, *x* must be a certified doctor in some health organization *org*, and furthermore the organization must be a certified health organization

- Policy (rule):

  The credential must be acquired from a foreign domain *org*

  canActivate(*x*, Doc(org))←*auth*.canActivate(*x*, CertDoc(*org*)),

  org@auth.canActivate(*org*, CertHealthOrg()), *auth*∈RegAuthorities()

- Scenario:

  A health organization, Addenbrooke's Hospital (*z*), is willing to reveal its CertHealthOrg credential to *x*, signed by the registration authority of East England (*y*), if *x* belongs to EHR servers

- Policy (rule):

  canReqCred(*x*, *y*.canActivate(*z*, CertHealthOrg())←

  x@*auth*.canActivate(*x*, CertEHRServ()),

  y=RegAuthEastEngland ∧ *z*=Addenbrookes, *auth*∈RegAuthorities()

- Whether *x* is willing to reveal her credential of EHR server might be further restricted by *x*'s canReqCred() policy. Therefore, the trust negotiation phase is triggered in Cassandra "almost for free"

Queries in Cassandra

- The queries in Cassandra takes the same form as credential, and the answers to the query are a set of constraints

$$E_{loc} \, @ \, E_{iss} . p_0 (\vec{e_0}) \leftarrow c$$

- Examples:

UCam@UCam.canActivate(*x*, Student(*subj*))←*subj*=Maths
might return {*x*=Alice, *x*=Bob}

UCam@UCam.canActivate(*x*, Student(*subj*))←*subj*=Maths, *x*=Alice
will simply return {true}

Deduction and Evaluation

- Given the syntax and semantics of:

  query, rule, credential, and predicate,

  how to evaluate a query based on the policy becomes a pure logic-reasoning problem in Datalog_C
- Top-down vs. Bottom-up

  - Bottom-up is not suitable, goal-oriented (top-down) is desirable
- Termination

  - standard SLD top-down algorithm may run into infinite loops
  - Cassandra uses a modified version of Toman's memoing algorithm
  - Constraint compactness is a sufficient condition on constraint domains to guarantee a finite and computable semantics for any finite global policy set *P*.

Strengths

- A complete system
  - not only a specification language
  - also consider trust negotiation
- Tunable expressiveness with constraint
  - constraint domain *C* is not a part of the language, and its definition can be integrated into Policy Evaluator module
- Language construct is small
  - clear,
  - easy to understand
  - easy to specify

Weaknesses

- ***Discussion…***

Weaknesses

- Some policies are hard to specify
  - e.g. Separation of duty
  - much harder to read and understand than the traditional representations
- Hard to claim "tunable"
  - need to explicitly enumerate the policies it supports one by one
  - impossible to "formally prove"
- Poses many restrictions on implementation
  - has to based on Datalog_C
  - no perfect algorithm to do the reasoning