OASIS

An Open Architecture for Secure Interworking Services

Richard Hayton
Citrix Systems (Cambridge)

Ken Moody
University of Cambridge

Access Control Fundamentals

- Access control is about classifying clients
  - those that may perform some action
  - those that may not
- If we wish to support transfer of privileges
  - we must also classify clients that may delegate access
  - and those that may be delegated to…...
- We *could* consider transfer of privilege as the right to modify the access control policy
  - the standard ACL approach
  - However this is dangerously difficult to control

An alternative approach

- Formalise the rules by which the *dynamic aspect* of a policy may evolve
  - i.e. who may delegate, revoke etc.
  - These rules must allow recursive specification
    - X may delegate R to Y, but Y may only delegate R’ to Z
- Describe delegation in terms of the *roles* of the clients
  - easier to understand than delegation of rights
  - easy to specify recursive cases
- This approach distinguishes between *static* policy rules and *dynamic* policy instantiations.

Why separate STATIC and DYNAMIC aspects?

- static policy can be analysed to see if a particular type of access is feasible
  - "is it possible for a student to see the exam"
  - there are decidability issues for large systems

- It is not possible to grant access that breaches policy
  - e.g. accidentally (mis)editing an ACL
  - all valid applications of policy rules are embodied in the static policy
- Easier to manage

RDL: Role Definition Language

- A means of expressing static policy statements
  - Described in terms of the preconditions for role entry
  - Strong grounding in formal methods
  - Extensions for delegation and revocation

- Policies as Proofs
  - STATIC Policy corresponds to a set of axioms
  - DYNAMIC Policy corresponds to a set of theorems
  - Validating access ≡ validating a proof

An RDL Statement

- grant client rolename($args_1$) ← if client already has rolename($args_2$)
  [∧ rolename($args_3$)...]
  [**:** constraints on args]

- e.g.

  ChiefExaminer() ← LoggedOn(ajh,*x*)

  : *x* in TrustedServers

Delegation

- rolename(args) ← rolename(args) ♋ rolename(args)
  [**:** constraints on args]

- e.g.

  Examiner(*e*) ← LoggedOn(*p*, *x*)
  ♋ ChiefExaminer() **:** *p* in Staff

RDL Statements as Horn Clauses

Examiner(*e*) ← LoggedOn(*p*, *x*)

ꝏ ChiefExaminer() **:** *p* in Staff

Is equivalent to

α requests Examiner(e) based on L,D,C,{E}

L is of the form "α has LoggedOn(p,x)"

D is of the form "β delegates Examiner(e) to P"

"α has P" is provable from {L,E}

C is of the form "β has ChiefExaminer()"

p is a member of the set Staff

α may be issued with "α has Examiner(e)"

Revocation

- May want to revoke for a variety of reasons
  - delegator wishes to revoke
  - delegator is revoked
  - preconditions fail (e.g. no longer logged on)
  - side conditions fail (e.g. removed from a group)
- Must control which of these are *active*
  - e.g. cannot revoke a vote
- Mechanism for revocation should be rapid

Specifying Revocation

- Clauses so far are ENTRY CONDITIONS
- Must be true to allow entry to a role
- Wish to revoke when some conditions no longer hold...
  - MEMBERSHIP RULES
- Revocation occurs when membership can no longer be proved using the MEMBERSHIP RULES

Revocation Example

- Revoke Candidate role
  - at the whim of the Examiner
  - if the Examiner is revoked
  - if the candidate ceases to be a student
- Don’t revoke under any other circumstances
- Specify by annotating RDL statement

Annotated RDL for Revocation

Candidate(p,e) $\leftarrow$ LoggedOn(p,x)

$\circ\!\!\circ$* Examiner(e)*

: (p in Students)*

An example theorem:

C_2 has Examiner(Math)

C_2 delegates Candidate(Fred,Math) to C_1

Fred in Students

C_1 has Candidate(Fred,Math)

Policy Definition : Summary

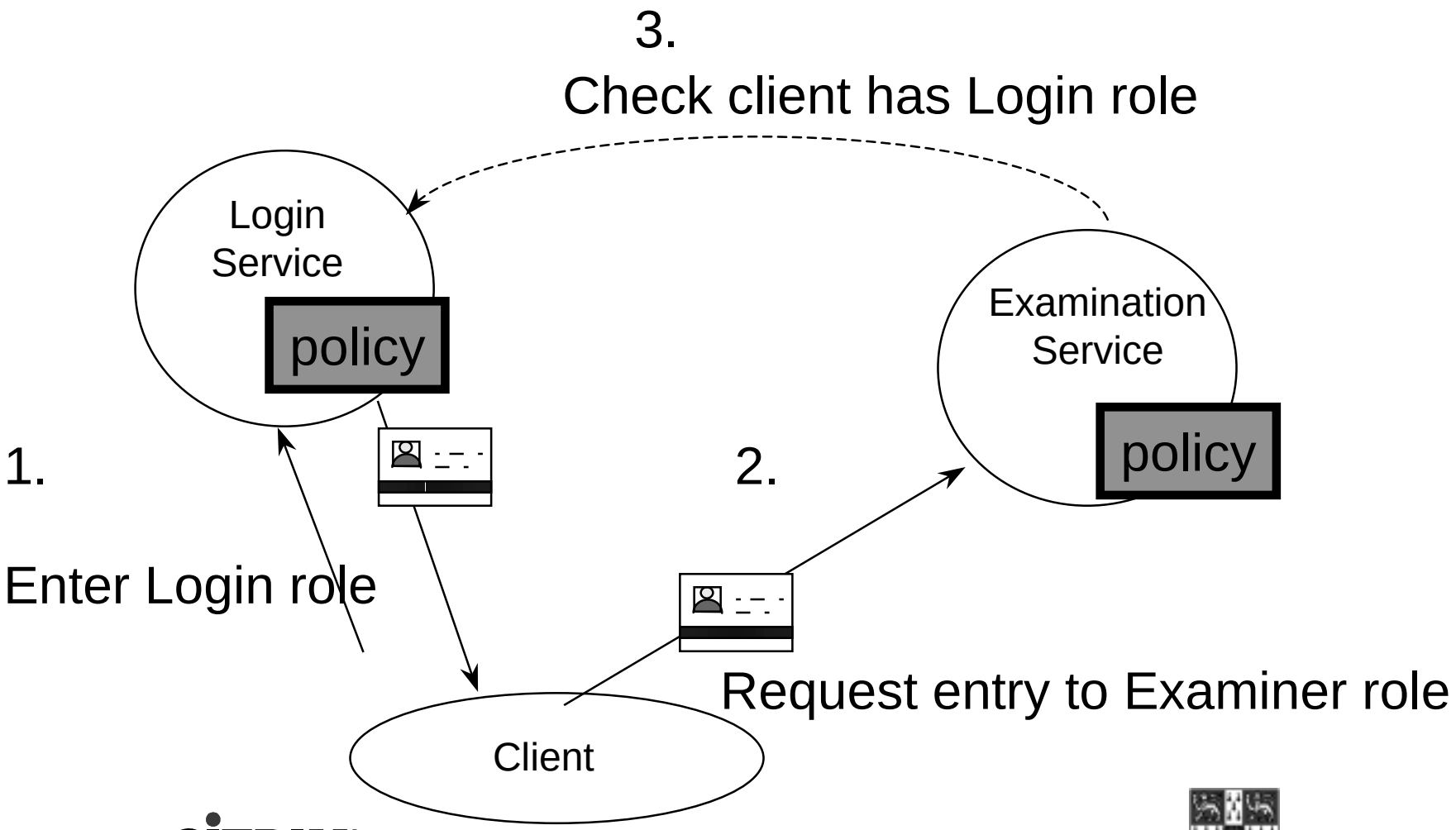
- We must distinguish static from dynamic policy
  - the only effective way to manage complexity
- RDL provides straightforward but powerful policy expression
  - formal grounding
  - extensible

- *so how does it all work in practice?*

Implementation Approach

- Services
  - manage policy related to service objects and service roles
  - issue certificates to clients to represent entry to a role
- Clients
  - obtain certificates from a variety of services
  - choice which certificate to use when
- Authentication
  - integrity check on client specific certificate
  - validation of proof of certificate theorem
    - involves a callback to the issuing service (to allow rapid revocation)
    - we build proofs when certificates are issued to make this fast

One service as a client of another



Policy Separation

- A service may make use of the roles issues by another service *without* being concerned with the mechanisms by which these roles are issued and revoked.

- This is a powerful abstraction
  - allows us to encapsulate legacy systems
  - allows for system evolution
  - new services do not weaken old services
  - services can be separately administered
  - different services can make different security/efficiency/availability trade-offs.

all the advantages of object oriented programming

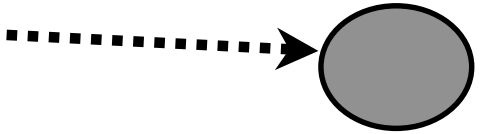
Managing Proofs

- We construct proofs dynamically, as certificates are issued
  - this allows rapid validation of role membership
  - These proofs may be collapsed due to revocation
- The implementation of this is the key to OASIS
  - the rest of the talk will give an overview of how this works
  - We have tested this with a prototype implementation

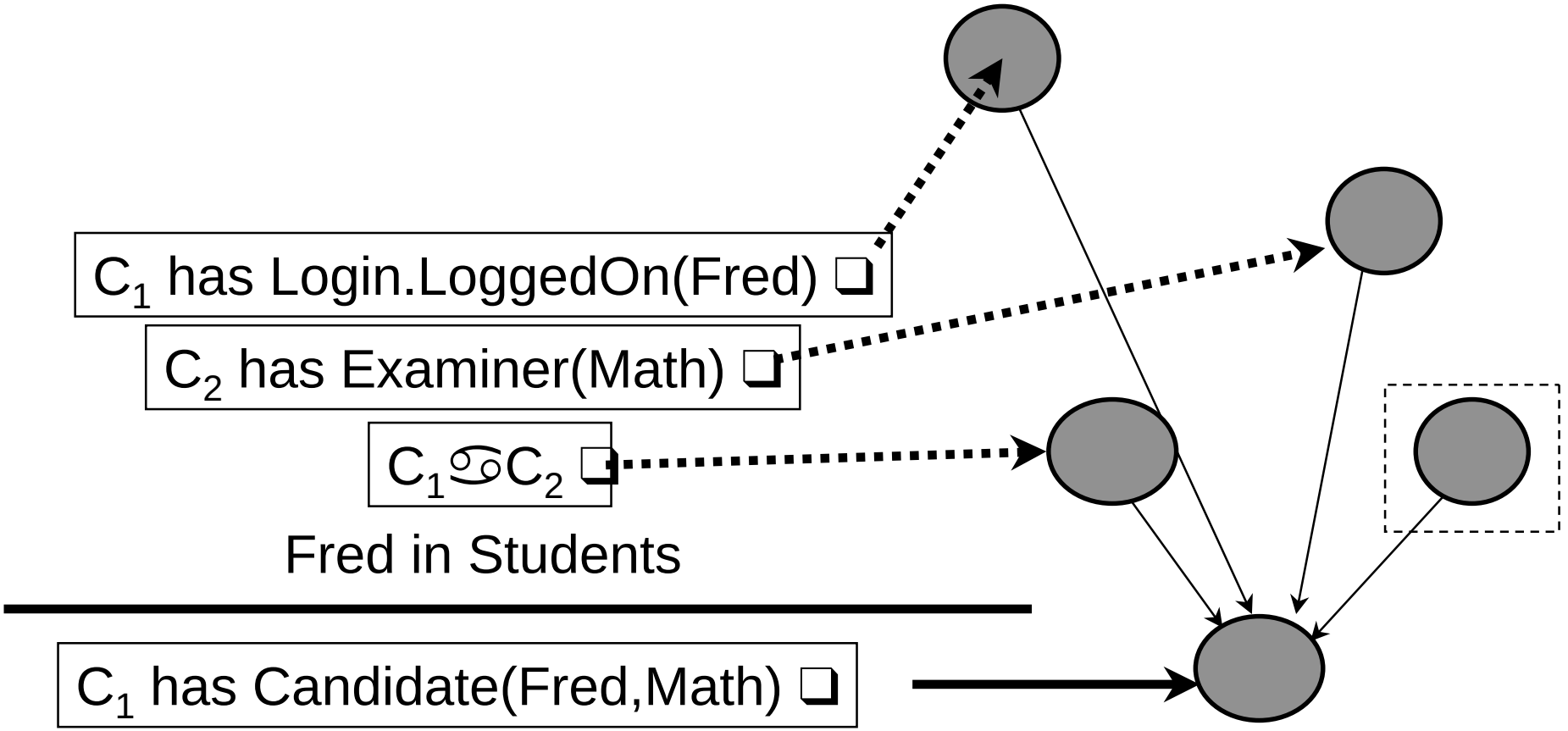
Managing Proofs - Credential Records

- Small records linked to form a proof tree
  - Each record represents this service's belief about some fact
    - validity of a role membership
    - group membership
    - other belief (e.g. it is Sunday)
- Each certificate is validated by a single credential record
    - fast to validate
    - easy to construct
    - easy to revoke

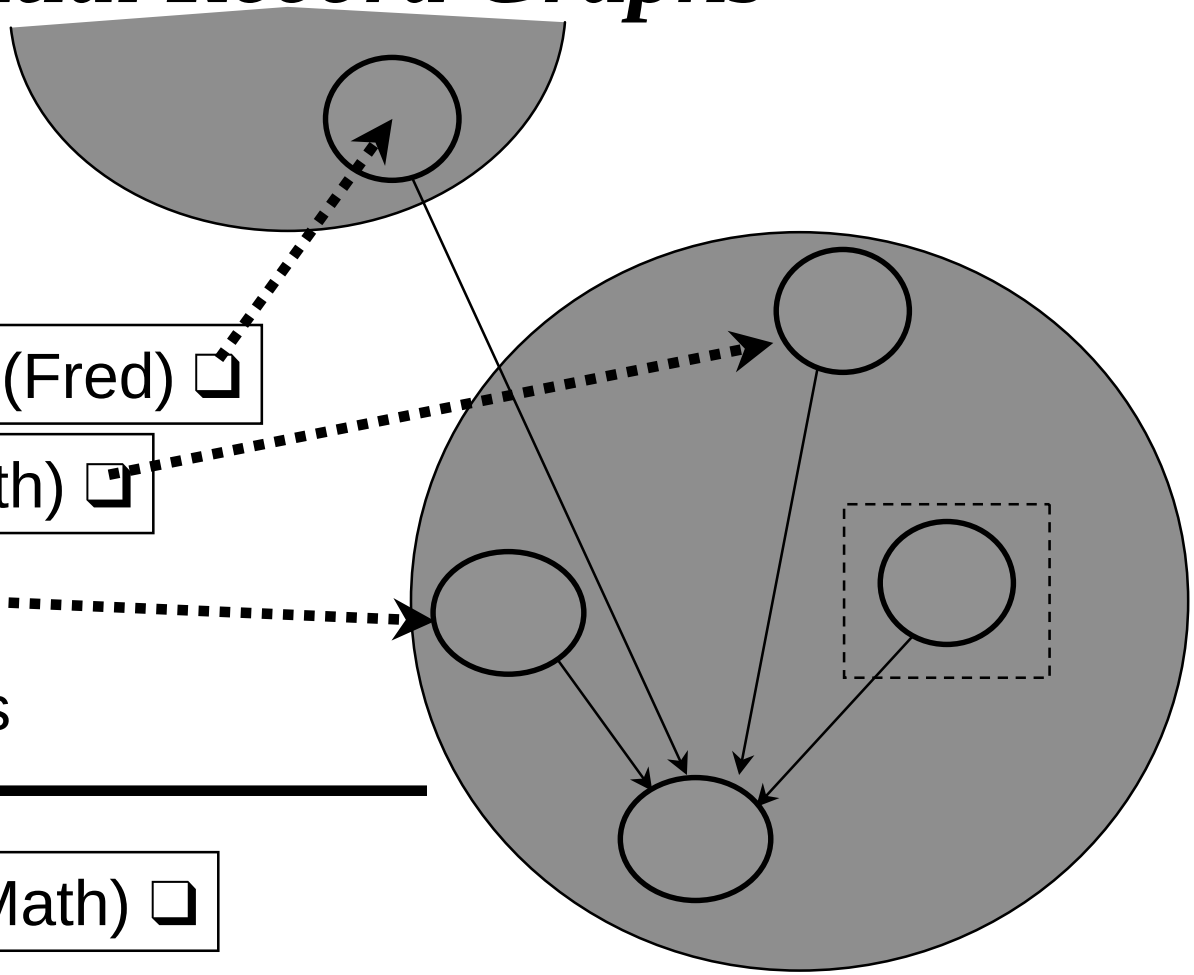
C_1 has Candidate(Fred,Math) ❑



Credential Record Graphs



Credential Record Graphs



C_1 has Login.LoggedOn(Fred) ❑

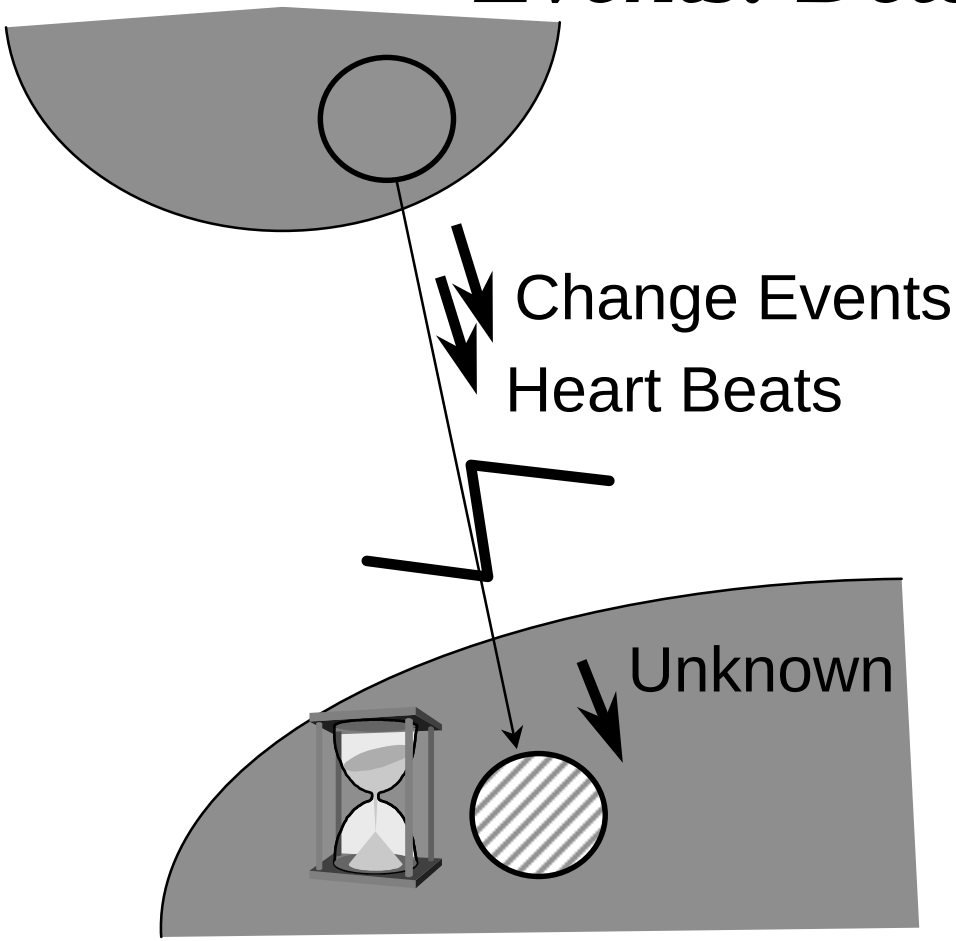
C_2 has Examiner(Math) ❑

$C_1 \bowtie C_2$ ❑

Fred in Students

C_1 has Candidate(Fred,Math) ❑

Events: Detecting Failure



- *on a failure*
  - mark records as unknown
  - control use of related certificates
  - don’t destroy state
    - rapid recovery
- Tuneable implementation
  - heartbeat interval
  - acceptable delay before signalling failure

Summary

- Policy Definition
  - Need a powerful language for STATIC policy
  - Need a mechanism for applying policy DYNAMICALLY
- Scale & Complexity
  - Each service manages its own roles
  - Implementation(policy) is encapsulated in service
    - expression, mechanism, trade-offs …..
- Validation
  - dynamically build proof trees for rapid validation
  - rapid, selective revocation is then straightforward