Fast Distributed Evaluation of Stateful Attribute-Based Access Control Policies

Thang Bui, Scott D. Stoller, Shikhar Sharma

▶ **To cite this version:**

Thang Bui, Scott D. Stoller, Shikhar Sharma. Fast Distributed Evaluation of Stateful Attribute-Based Access Control Policies. 31th IFIP Annual Conference on Data and Applications Security and Privacy (DBSEC), Jul 2017, Philadelphia, PA, United States. pp.101-119, 10.1007/978-3-319-61176-1_6 . hal-01684356

HAL Id: hal-01684356

https://inria.hal.science/hal-01684356v1

Submitted on 15 Jan 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Fast Distributed Evaluation of Stateful Attribute-Based Access Control Policies*

Thang Bui, Scott D. Stoller, and Shikhar Sharma

Department of Computer Science, Stony Brook University, USA

Abstract. Separation of access control logic from other components of applications facilitates uniform enforcement of policies across applications in enterprise systems. This approach is popular in attribute-based access control (ABAC) systems and is embodied in the XACML standard. For this approach to be practical in an enterprise system, the access control decision engine must be scalable, able to quickly respond to access control requests from many concurrently running applications. This is especially challenging for stateful (also called history-based) access control policies, in which access control requests may trigger state updates. This paper presents an policy evaluation algorithm for stateful ABAC policies that achieves high throughput by distributed processing, using a specialized multi-version concurrency control scheme to deal with possibly conflicting concurrent updates. The algorithm is especially designed to achieve low latency, by minimizing the number of messages on the critical path of each access control decision.

1 Introduction

Separation of access control logic from other components of applications facilitates uniform enforcement of policies across applications in enterprise systems. This approach is adopted in the ISO standard for access control in open systems [13] and the XACML standard[1]. Servers that run the access control policy evaluation algorithm and provide access control decisions to applications are called *policy decision points* (PDPs) in XACML terminology. In this paper, we refer to them simply as *servers*, since we do not discuss other kinds of server.

For this approach to be practical in an enterprise system, the policy evaluation algorithm must be scalable, able to quickly respond to access control requests from many concurrently running applications. To scale beyond the capacity of a single server, distributed policy evaluation algorithms are needed, to coordinate concurrent processing of requests on multiple servers. This is relatively straightforward if the policy and the information it references are static.

* This material is based on work supported in part by NSF Grants CNS-1421893, and CCF-1414078, ONR Grant N00014-15-1-2208, AFOSR Grant FA9550-14-1-0261, and DARPA Contract FA8650-15-C-7561. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

[1] `http://www.oasis-open.org/committees/xacml/`

However, this is challenging for *stateful* (also called *state-modifying*, *dynamic*, or *history-based*) access control policies, in which access control requests may trigger state updates, i.e., updates to information referenced by the policy. The classical examples of stateful access control policies are dynamic separation-of-duty (DSOD) policies, such as the Chinese wall policy [5] and DSOD in role-based access control (RBAC) [2]. Another classic category of stateful access control policies are usage control policies [20], such as policies that limit the number of times a user can view a video or the number of videos that a user with a particular type of subscription can view each month. The research literature contains numerous additional examples of stateful access control policies, policy models, and policy evaluation algorithms [10,11,6,3,4,17,12,19,14,8,22,18]. In the context of Attribute-Based Access Control (ABAC), the updated state is typically attribute data.

The main challenge in distributed policy evaluation algorithms for stateful policies is ensuring serializability, as in concurrent transaction processing in databases [21]. Processing of each access control request, including its reads of attribute data and its updates to attribute data, should be serializable with respect to processing of other requests. Since concurrent requests may read or write the same attribute data, a concurrency control mechanism is needed to ensure this.

To illustrate the importance of serializability in this context, consider a typical Chinese wall policy in which companies A and B are in the same conflict of interest (COI) class, so user who has accessed documents of one them cannot access documents of the other. When a server allows a request for access to documents of either company, it updates a user attribute to reflect this. Suppose a devious user concurrently submits an access request for a document of company A to one server, and an access request for a document of company B to another server. In a non-serializable execution in which both requests are evaluated in the initial state (where the user has not accessed any documents), both requests could be permitted, violating the intended policy. In a serializable execution, the result must be equivalent to a serial execution, where one of the requests sees the effect of the update performed by the other request, causing the second request to be denied, as it should be.

A straightforward approach to this problem is to use a distributed replicated database that supports serializability for multi-row transactions, and to evaluate each request in a transaction. However, this requirement eliminates well-known scalable NoSQL databases, such as Bigtable [7], Cassandra[2], and MongoDB[3], which achieve scalability in part by supporting only single-row transactions. Master-slave replication in SQL databases, such as MySQL[4], allows multi-row transactions, but has limited scalability, because all read-write transactions must be submitted to a single master server, and provides inadequate consistency guarantees, because slaves can return slightly out-of-date data. Multi-phase commit

[2] `http://cassandra.apache.org/`
[3] `https://www.mongodb.com/`
[4] `https://dev.mysql.com/`

protocols, such as in Oracle, IBM DB2, and Microsoft SQL Sever, allow multi-row tranasctions and ensure serializability, but are less scalable.

Decat *et al.* present a distributed policy evaluation algorithm for stateful ABAC policies [8] that is more scalable than multi-phase commit protocols, by exploiting the fact that evaluation of an ABAC request involves at most two objects (i.e., two rows), typically called the *subject* and the *resource*. Their algorithm uses a specialized scheme for optimistic concurrency control [21, Section 15.5]. Their experimental results demonstrate that their algorithm scales well in terms of throughput. However, their algorithm incurs a significant increase in latency, since processing of each request involves a chain of 6 messages (including the messages to and from the client).

This paper presents a new distributed policy evaluation algorithm for stateful ABAC policies. The algorithm is called FACADE (Fast Access Control Algorithm with Distributed Evaluation). It uses a specialized scheme for multiversion timestamp ordering concurrency control [21, Section 15.6] that simultaneously achieves low latency by minimizing the length of the message chain on the critical path (i.e., the message chain ending with the result sent to the client). Low latency is of obvious importance for interactive applications: developers struggle to keep the latency of the application's core functionality within limits acceptable to users, especially for multi-tier enterprise applications, where many requests involve processing by multiple servers (web servers, application servers, database servers, etc.), and latency contributions from non-core functionality such as access control are acceptable only if they are low. Low latency is also important for batch applications. These applications often process large amounts of data, hence requiring many access control checks. If the latency of these checks is not kept very low, the repeated delays in the core application processing will cause poor system utilization. Reducing the number of messages per request has the additional benefit of reducing the required network capacity.

FACADE processes read-only requests differently than read-write requests, in contrast to Decat *et al.*'s algorithm, which processes all requests the same way. This, together with use of multiversion timestamp ordering concurrency control, enables FACADE to use especially short message chains for read-only requests. Multiversion timestamp ordering concurrency control has the desirable property that *read-only requests never abort*. This helps FACADE use a shorter critical path than Decat *et al.*'s algorithm for read-only requests.

FACADE also uses shorter message chains than Decat *et al.*'s algorithm for read-write requests. This is achieved partly by use of multiversion concurrency control and partly by specialization to requests that update at most one object. This specialization is motivated by the observation that in every stateful policy given as an example in every paper cited above, each request updates the state of at most one object. FACADE can be extended to handle requests that update two objects, but that extension is not described in this paper.

FACADE is more flexible than Decat *et al.*'s algorithm, in that FACADE allows an object to be a subject and a resource, while Decat *et al.*'s algorithm requires the sets of subjects and resources to be disjoint [8, Section 3.4]

We ran experiments, described in Section 3, to compare the performance of FACADE and Decat *et al.*'s algorithm. Our experiments show that FACADE has significantly lower average latency, uses significantly fewer network messages per request, and has slightly higher throughput than Decat *et al.*'s algorithm in many cases.

2 Algorithm

System Architecture. We adopt the system architecture in [8]. There are two types of hosts: *clients* and *servers*. Each client runs applications and a small client-side stub that interacts with the access control servers. Each server runs three kinds of processes: a *coordinator*, which receives requests from clients and is responsible for concurrency control, a *database*, which stores a copy of the attribute data used by the policy, and one or more *workers*, which evaluate requests based on the access control policy and send the result to the coordinator and/or client.

Each worker reads attribute data from the co-located replica of the database. Workers never update the database. The set of objects is partitioned across the set of coordinators. Thus, for each object x, there is a unique coordinator, denoted coord(x), responsible for x; we also say that coord(x) *manages* x. Only coord(x) submits updates of x to the master database (this is done using a standard database connector, such as ODBC or JDBC, regardless of whether the master database is on the same server or a different server). Coordinators never read the database.

Multiversion timestamp ordering concurrency control. Before presenting our algorithm, we briefly review multiversion timestamp ordering concurrency control [21, Section 15.6], with a change in terminology: we refer to "requests" in place of "transactions". A sequence of *versions* is associated with each data item. In FACADE, we treat each attribute of each object as a data item. Each version v has a value v.value, a write timestamp v.wts (the timestamp of the request that created v), and a read timestamp v.rts (the largest timestamp of any request that successfully read v). Each request req is assigned a timestamp req.ts. Let v denote the most recent version of x.attr whose timestamp is at most req.ts. A read of x.attr by req returns v.value. A write by req requires a conflict check: if req.ts $<$ v.rts, then req aborts and restarts; if req.ts $==$ v.wts, then the value of v is overwritten; otherwise (if req.ts $>$ v.rts), a new version of x.attr is created. Note that reads never cause aborts, and read-only transactions always commit.

To support conflict checking, each coordinator maintains a data structure containing the read timestamp and write timestamp of every version of an attribute created during the coordinator's current session (i.e., since the coordinator process started running). This data structure does not store the value of each version, since it is not needed for conflict checking. Entries for old versions can be garbage-collected; details are straightforward and omitted. Although this data structure has some information overlap with cachedUpdates, we keep the two data structures separate for clarity, because they serve different purposes.

This data structure is accessed using two functions. getVersion(x,attr,ts) returns the most recent version of x.attr written at or before ts; if no such version exists, it returns a special version v with v.wts=0 and v.rts=0, representing the last version written in the previous session (any timestamp guaranteed to precede all timestamps generated in the current session is safe; 0 is a convenient choice). addVersion(x,attr,ts) creates and stores a version of x.attr with write timestamp and read timestamp equal to ts.

Database. To avoid use of a heavyweight multi-phase commit protocol in the database, we assume a database that supports master-slave (also called primary-secondary) replication, in which updates are committed at one replica, called the *master* or *primary*, and the updates are visible at the other replicas, called *slaves* or *secondaries*, within a known time limit, called the *database latency*. This assumption is satisfied by the replication schemes in popular databases, such as primary-secondary replication in MongoDB and master-slave replication in MySQL. Loose bounds on the database latency are sufficient: the size of the database latency has little effect on FACADE's performance, mainly affecting how long updates are cached by coordinators. Since distributed concurrency control is provided by the coordinators, it does not matter what, if any, centralized concurrency control scheme is used by the master replica of the database.

FACADE masks the database latency in the same way as Decat *et al.*'s algorithm. Each coordinator maintains a LRU cache of recent committed updates to objects it manages, and it piggybacks on each request (when forwarding the request to a coordinator or worker) the cached updates for objects it manages that are involved in the request. Each cached update specifies a write timestamp as well as an attribute and its new value. A cached update is never evicted before the current time exceeds the update's write timestamp plus the database latency. The cache is accessed using the function cachedUpdates(x), which returns the set of cached updates to x.

FACADE needs to store multiple versions of objects in the database. This can easily be done in any database, by including a "version" column in the database schema. Our implementation using MySQL works this way.

Request objects. We model requests as objects with fields subject, resource, ts (timestamp), cachedUpdates[i] (i=1 and i=2 for piggybacked cached updates to subject and resource, respectively), worker (worker selected to evaluate this request), and evalResult (result of evaluating the request, described below).

Policy language. FACADE is independent of the details of the policy language. Any ABAC policy language can be used, provided it can express updates. For example, XACML can be used, with updates expressed as obligations, as in [20,8]. Details of the policy language are abstracted behind an interface containing a single function evaluateRequest(policy,request) that returns an EvalResult object with these fields: decision (permit or deny), readAttr[i] (i=1 and i=2 for the set of attributes of the subject and resource, respectively, read during evaluation

of the request), updatedObj (the index of the updated object, if any, otherwise -1), rdonlyObj (if updatedObj > 0, this is the index of the other object, otherwise -1), and updates (set of attribute-value pairs, specifying updates to updatedObj). The index values are interpreted as: 1=subject, 2=resource. evaluateRequest evaluates the request using attribute values current as of req.ts, reading values from req.cachedUpdates when they exist, otherwise reading values from the database using queries with timestamp req.ts.

Bounds on attribute accesses. Our algorithm can exploit bounds on attribute read and written by requests, when available, to improve performance. In particular, for a request r, for each object x that might be accessed by r (namely, the subject and resource), the client stub provides (1) a lower bound on the set of attributes of x that will definitely be read by r, (2) an upper bound on the set of attributes of x that might be read by r, and (3) an upper bound on the set of attributes of x that might be updated by the request. It is always safe to use the trivial bounds, i.e., the empty set for (1) and the set of all attributes for (2) and (3). When tighter bounds are available for (1), the algorithm can sometimes use them to conclude that a request definitely conflicts with an in-progress request r, without waiting to learn the exact set of attributes read by r. When tighter bounds are available for (2) and (3), the algorithm can sometimes use them to conclude that two requests involving the same object access disjoint sets of attributes and hence cannot conflict, without waiting to learn the exact sets of attributes they accessed. Note that these situations arise only in the typically small fraction of cases that two concurrent requests access the same object, and at least one of the requests is not known to be read-only.

Tighter bounds can often be obtained from basic knowledge about the request and the policy. The code or rules defining these bounds could be written manually for small systems or generated by a straightforward static analysis of the access control policy, based on the types of object and type of action in each rule and the names of the attributes read and written by each rule. For example, consider an access control system for an online video service, in which requests to play a video are subject to usage control to limit the number of views, and all other requests (browsing the video catalog, paying for a video, account maintenance, etc.) are not. In this system, a client can identify a request as read-only if the resource type is not "video" or the action is not "play".

These bounds are provided by defining (possibly using trivial bounds) the following policy-specific functions, where x is req.subject or req.resource.

- defReadAttr(x, req) is a set of attributes of x definitely read by req.
- mightReadAttr(x, req) is an upper bound on the set of attributes of x that might be read by req (including definitely read attributes).
- mightWriteAttr(x, req) is an upper bound on the set of attributes of x that might be updated by req.

Sequence Diagrams. We give brief overviews of Decat *et al.*'s algorithm and our algorithm, focusing on the message patterns shown in the sequence diagrams in

Figure 1. The sequence diagrams show the common case in which the request does not restart due to a conflict and the two objects accessed by the request are managed by coordinators on different servers. Accesses to the database are not shown; they are the same for Decat *et al.*'s algorithm and FACADE.

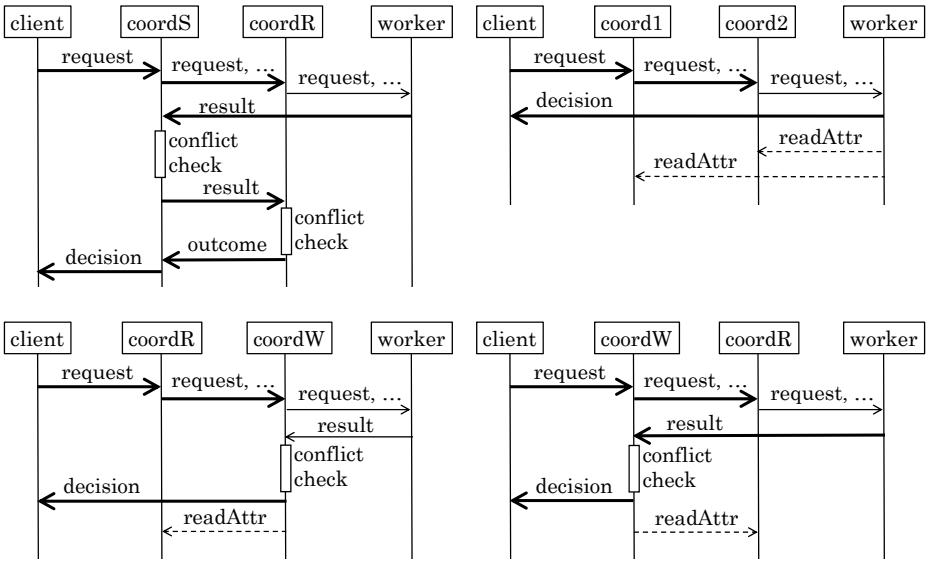

Fig. 1. Sequence diagrams. Top left: Decat *et al.*'s algorithm. Top right: FACADE for read-only request. Thick and thin solid lines are non-local and local messages, respectively, on the critical path. Dashed lines are messages not on the critical path. Bottom left: FACADE for read-write request, when client correctly predicts a read-only object. Bottom right: FACADE for read-write request, when client incorrectly predicts a read-only object.

Overview of Decat et al.'s algorithm. In Decat *et al.*'s algorithm, the client sends the request to coordS, the coordinator for the subject of the request. coordS updates data structures used for conflict detection and then forwards the request (with piggybacked cached committed updates) to coordR, the coordinator for the resource of the request, which does the same and then forwards to the request to a worker on the same server. The worker evaluates the request and then sends the result to coordS. coordS checks for conflicts involving the subject; specifically, it checks whether any attribute of the subject read by the request was updated after it forwarded the request to coordR (any such update was not piggybacked on the request and hence might not have been used in its evaluation). If there is no conflict, it forwards the result to coordR, which performs a similar conflict check and, if there is no conflict, commits the updates (if any) to the resource, and then sends the outcome of the conflict check to coordS. coordS commits the updates to the subject and then sends the decision to the client. If either

coordinator detects a conflict, the request is restarted. After coordS sends the result to coordR and before it receives the outcome of coordR's conflict check, it treats the request's updates to the subject specially, as *tentative updates*; for details, see [8].

Overview of FACADE for read-only requests. The client sends the request to coord1, the coordinator for one of the objects accessed by the request (either one is fine). coord1 updates data structures used for conflict detection and then forwards the request (with piggybacked cached committed updates) to coord2, the coordinator for the other object accessed by the request. coord2 updates its data structures and forwards the request to the worker. The worker evaluates the request, sends the decision to the client, and sends the sets of read attributes of the subject and resource to their respective coordinators, which update the read timestamps of the read versions. It is safe for the worker to send the decision directly to the client, because read-only requests never abort in FACADE.

Note that this message pattern is used for any request that turns out to be read-only, regardless of whether this is known in advance, i.e., regardless of whether mightWriteAttr is empty for either object involved in the request.

Overview of FACADE for read-write requests. When the client sends the request to the coordinator for an object not updated by the request, we say that the client *correctly predicts a read-only object* for the request. This is guaranteed if mightWriteAttr returns an empty set for at least one object involved in the request, and has 50% probability otherwise. It is preferable for the client to send the request to such a coordinator, denoted coordR, because the worker sends the evaluation result to the coordinator for the updated object, denoted coordW, and that result message is local if the worker is co-located with coordW, which happens if coordR receives the request from the client and forwards it to coordW. If mightWriteAttr returns a non-empty set for both objects, then the client arbitrarily selects a coordinator to which to send the request. If that turns out to be coordW, we say that the client *incorrectly predicts a read-only object* for the request. The only consequence is that the worker's result message is a network message instead of a local message.

When the client correctly predicts a read-only object for the request, the client sends the request to the coordinator for that object, denoted coordR. coordR updates data structures used for conflict detection and then forwards the request (with piggybacked cached committed updates) to coordW. The worker evaluates the request and sends the result, including the decision and the sets of read and written attributes of the subject and resource, to coordW. coordW checks for conflicts; specifically, it checks whether any attribute updated by this request was read by a request with a later timestamp. Even if there is no conflict yet, a conflict could arise later, involving a request with a later timestamp that has already been forwarded and might read the attribute. A set of such requests, called "pending might read requests", is associated with each version of an attribute. The worker waits until there are no such pending might read requests and then checks for conflicts again. If there is no conflict, it commits

the updates, sends the decision to the client, and sends the set of read attributes of the other object to the other coordinator.

When the client incorrectly predicts a read-only object for the request, the message pattern is the same, except that coordW receives the request first and then forwards it to coordR, and the evaluation result message from the worker to coordW is a network message instead of a local message.

Handling of requests known to be read-only. A request req is *known to be read-only* iff mightWriteAttr(req.subject, req) and mightWriteAttr(req.resource, req) are empty. Handling of requests known to be read-only is described separately from handling of other requests, for ease of understanding, although the two are similar in places, and the code for them is integrated in our implementation. Handling of requests known to be read-only follows the pseudocode in Figure 2. The pseudocode syntax is generally Python-like, except we denote tuples using angle brackets instead of parentheses. Implicitly, coarse-grain locking is used to ensure that coordinators process each incoming message atomically, i.e., without interruption by processing of other messages (as an optimization, finer-grained locking could be used).

Handling of read-write requests. Handling of other requests follows the pseudocode in Figures 3 and 4.

Liveness. The algorithm presented in the pseudocode is deadlock-free: the inequality on timestamps in the **await** statement in Figure 4 ensures that two requests cannot be stuck waiting for each other. However, it can starve some read-write requests. For example, a long stream of reads to an attribute x.attr can cause the condition in the **await** statement in Figure 4 to remain true for a long time, causing a pending update to x.attr to starve. The underlying reason is that FACADE gives precedence to reads over writes, in the sense that reads never abort, and writes can be aborted due to conflicting reads.

To counter-balance this, and thereby help prevent starvation of writes, we modify the algorithm to delay reads in two cases (these modifications are not reflected in the pseudocode). (1) After a coordinator c receives ⟨"request", req, 1⟩ from a client, if req might update req.obj[1], c delays processing of incoming requests that potentially conflict with req (temporarily storing them in a queue) until c determines the outcome (commit or restart) of the current execution of req, at which time c processes the delayed requests normally. An incoming request req2 potentially conflicts with req if req2 might read an attribute that req might update. (2) After a coordinator c receives an evaluation result message ⟨"result", req⟩ that includes updates to an object x managed by c, while c is waiting for the **await** condition to become true, c delays processing of incoming requests that potentially conflict with those updates until c determines the outcome (commit or restart) for req, at which time c processes the delayed requests normally An incoming request req2 potentially conflicts with the updates if req2 might read one of the updated attributes. Note that these two kinds of delays

```
1. client:
# for read-only requests, the coordinator order (subject first or resource first) does
# not affect correctness or performance. arbitrarily do subject first.
req.obj[1], req.obj[2] = req.subject. req.resource
send ⟨"request", req, 1⟩ to coord(req.obj[1])

2. coordinator: on receiving ⟨"request", req, 1⟩:
x = req.obj[1]
req.ts = now()    # now() returns the current date-time.
for attr in defReadAttr(x, req):
  getVersion(x, attr, req.ts).rts = req.ts
for attr in mightReadAttr(x, req) - defReadAttr(x, req):
  getVersion(x, attr, req.ts).pendingMightReads.add(⟨req.id, req.ts⟩)
req.cachedUpdates[1] = cachedUpdates(x)
send ⟨"request", req, 2⟩ to coord(req.obj[2])

3. coordinator: on receiving ⟨"request", req, 2⟩:
x = req.obj[2]
for attr in defReadAttr(x, req):
  getVersion(x, attr, req.ts).rts = req.ts
for attr in mightReadAttr(x, req) - defReadAttr(x, req):
  getVersion(x, attr, req.ts).pendingMightReads.add(⟨req.id, req.ts⟩)
select worker w to evaluate this request
req.worker = w
req.cachedUpdates[2] = cachedUpdates(x)
send req to w

4. worker: on receiving req:
req.evalResult = evaluateRequest(policy, req)
send ⟨"decision", req.id, evalResult.decision⟩ to req.client
send ⟨"readAttr", req, 1⟩ to coord(req.subject)
send ⟨"readAttr", req, 2⟩ to coord(req.resource)

5. coordinator: on receiving ⟨"readAttr", req, i⟩:
x = req.subject if i==1 else req.resource
for attr in mightReadAttr(x, req) - defReadAttr(x, req):
  v = getVersion(x, attr, req.ts)
  v.pendingMightReads.remove(⟨req.id, req.ts⟩)
  if attr in req.evalResult.readAttr[i]:
    v.rts = req.ts
```

Fig. 2. Handling of requests known to be read-only.

cannot lead to deadlock (i.e., to circular wait), because the delayed requests are younger than req.

Decat *et al.*'s algorithm can also starve requests. It gives precedence to writes over reads, in the sense that writes never abort, and reads can be aborted because of conflicting writes. Consequently, long streams of writes can starve read-only

```
1. client:
# if either object is known to be read-only, send req to its coordinator first.
if isEmpty(mightWriteAttr(req.obj[2], req)):
  req.obj[1], req.obj[2] = 2, 1
else:
  req.obj[1], req.obj[2] = 1, 2
send ⟨"request", req, 1⟩ to coord(req.obj[1])

2. coordinator: on receiving ⟨"request", req, 1⟩:
x = req.obj[1]
req.ts = now()    # now() returns the current date-time.
for attr in mightReadAttr(x, req)
  v = getVersion(x,attr,req.ts)
  v.pendingMightReads.add(⟨req.id, req.ts⟩)
req.cachedUpdates[1] = cachedUpdates(x)
send ⟨"request", req, 2⟩ to coord(req.obj[2])

3. coordinator: on receiving ⟨"request", req, 2⟩:
x = req.obj[2]
for attr in mightReadAttr(x,req)
  v = getVersion(x,attr,req.ts)
  v.pendingMightReads.add(⟨req.id, req.ts⟩)
select worker w to evaluate this request
req.worker = w
req.cachedUpdates[2] = cachedUpdates(x)
send req to w

4. worker: on receiving req:
req.evalResult = evaluateRequest(policy, req)
if req.updatedObj == -1:
  # req is read-only.
  send ⟨"decision", req.id, req.evalResult.decision⟩ to req.client
  send ⟨"readAttr", req, 1⟩ to coord(req.subject)
  send ⟨"readAttr", req, 2⟩ to coord(req.resource)
else:
  # req updated an object.
  send ⟨"result", req⟩ to coord(req.obj[req.updatedObj])
```

Fig. 3. Handling of requests not known to be read-only, part 1.

and read-write requests. Their algorithm does not incorporate any mechanism to compensate for this. This is probably acceptable for workloads in which writes are infrequent relative to reads.

Optimizations. Our implementation incorporates the following optimizations that are not reflected in the pseudocode. (1) If the same coordinator is responsible for both objects involved in a request, then the coordinator performs the processing for both objects together, without sending itself a message in between.

```
5. coordinator: on receiving ⟨"result", req⟩:
# req updates an object that this coordinator is responsible for.
x = req.obj[req.updatedObj]
conflict = checkForConflicts()
if not conflict:
  # wait for relevant pending reads to complete. await(expr) blocks until expr is true.
  await (∀⟨attr,val⟩∈req.updates. ∀⟨id,ts⟩∈getVersion(x, attr, req.ts).pendingMightReads.
          id == req.id or ts < req.ts)
  conflict = checkForConflicts()
  if not conflict:
    commit req.evalResult.updates to the database with write timestamp req.ts
    # cache the updates, and store the new versions for conflict checking
    for (attr, val) in req.evalResult.updates:
      cachedUpdates(x).add(⟨attr, val, req.ts⟩)
      addVersion(x,attr,req.ts)
    # update read timestamps
    for attr in mightReadAttr(x,req)
      v = getVersion(x,attr,req.ts)
      v.pendingMightReads.remove(⟨req.id, req.ts⟩)
      if attr in req.readAttr[req.updatedObj]:
        v.rts = req.ts
    # send decision to client
    send ⟨"decision", req.id, req.decision⟩ to req.client
    # send read attributes to coordinator for read-only object
    roCoord = coord(req.subject) if req.evalResult.rdonlyObj==1 else coord(req.resource)
    send ⟨"readAttr", req, req.evalResult.rdonlyObj⟩ to roCoord
  else:
    restart(req)
else:
  restart(req)

coordinator: on receiving ⟨"restart", req⟩:
remove entries for req from all pendingMightReads sets
restart processing of req, as if it were newly received from client

def checkForConflicts():
  for (attr, val) in req.updates:
    # note: if x.attr has not been read or written in this session, then
    # v is the special version with v.rts=0 and v.wts=0.
    v = getVersion(x, attr, req.ts)
    if v.rts > req.ts:
      return true
  return false

def restart(req):
  remove entries for req from all pendingMightReads sets
  # tell the other coordinator to restart processing of this request
  roCoord = coord(req.subject) if req.evalResult.rdonlyObj==1 else coord(req.resource)
  send ⟨"restart", req⟩ to roCoord
```

Fig. 4. Handling of requests not known to be read-only, part 2.

(2) The **await** statement in Figure 4 waits for all relevant pending reads to complete before checking whether any of them conflict with the pending update. As an optimization, when each relevant pending read completes, the coordinator immediately checks whether it conflicts with the pending update, and if so, immediately restarts the request performing the update. (3) To reduce the number of database queries, workers piggyback data read from the database on messages sent to coordinators, and coordinators add it to the data structure that caches recent committed updates. Note that caching of attribute data is done only by coordinators, not workers, because a coordinator performs all updates to objects it manages and hence knows when cached data is stale (relative to a specified request timestamp).

Fault-tolerance. Like Decat *et al.* in [8], we focus in this paper on scalability and leave detailed consideration of fault-tolerance for future work. We briefly sketch how to extend our algorithm to tolerate crash failures. A fault-monitoring service is needed to detect crashes and restart crashed processes. Requests that were in-progress at the time of a crash might be dropped. If a client does not receive a decision for a request in a reasonable amount of time, the client can re-submit the request with the same identifier. If the request is read-only, the worker simply re-evaluates it in the current state. If the request performs updates, the worker checks whether the request already committed, and if so, re-sends the original decision, otherwise re-evaluates the request in the current state. To support this, when a coordinator commits the attribute updates for a request, it also inserts a record containing the request id and decision in a request log table. The worker looks up the request id in this table before evaluating a request.

3 Evaluation

Implementation. We implemented FACADE in DistAlgo [16,15], an extension of Python with high-level communication and synchronization constructs. The DistAlgo compiler[5] translates DistAlgo into Python for execution. We also implemented Decat *et al.*'s algorithm in DistAlgo, to allow a performance comparison of the algorithms, not influenced by the performance of different programming language implementations (Decat *et al.*'s implementation is in Scala). Our implementations of both algorithms are publicly available[6]. The experimental platform consists of three desktop PCs with Intel Core 2 Quad processors (two at 2.83GHz, one at 2.66 GHz), with Gigabit Ethernet NICs connected to a Gigabit Ethernet switch, and running Windows 10 64-bit, Python 3.6, DistAlgo 1.0.9, and MySQL 5.7.17.

Workload. The workload consists of pseudorandom sequences of requests. The same seeds, hence the same workload, are used for corresponding experiments with the two algorithms. Configuration parameters for each experiment include:

[5] `http://sourceforge.net/projects/distalgo/files/`
[6] `http://www.cs.stonybrook.edu/~stoller/software/`

- nClient: number of clients. This is also the maximum number of concurrent requests, since each client sends a request and waits for the response before sending the next request.
- nWorker: number of workers per coordinator.
- nObj: number of objects in database. We use objects with 10 attributes, two of which are mutable (i.e., might be updated by access control policy rules).
- nRequest: total number of requests (split evenly among the clients)
- pWrite: probability that a request is read-write; other requests are read-only.
- pSameCoord: probability that the two objects involved in a request have the same coordinator. As discussed below, we emulate experiments with nCoord coordinators using our platform with 2 coordinators by setting pSameCoord=1/nCoord.
- wrongWrite: flag controlling accuracy of client's prediction of written objects. wrongWrite=0 means completely accurate. wrongWrite=1 means the prediction always includes an object not written by the request.
- wrongAttr: flag controlling accuracy of client's prediction of accessed attributes. wrongAttr=0 means completely accurate. wrongAttr=1 means the predictions of read and written attributes contain all attributes and all mutable attributes, respectively.

Latency. To evaluate how the performance, primarily latency, of FACADE would depend on the number of coordinators in a system, we ran experiments analogous to the latency experiments in [8, Section 3.4, Figure 9]. We use nClient=1, like they do, to measure the intrinsic latency of the algorithm, in the absence of contention. In their experiment, latency is measured instead as a function of the actual number of coordinators. However, the number of coordinators affects the latency only indirectly, by affecting the probability that the same coordinator is responsible for the two objects involved in the request. For clarity, we measure the latency directly as a function of this probability, by making pSameCoord a workload parameter, as described above. This also allows us to use a smaller platform for the experiments. Values of the other fixed workload parameters in these experiments are nWorker=1, nObj=1000, nRequest=5000 and pWrite=0.1. For FACADE, we repeat the experiments for each of the four possible combinations of values of wrongWrite and wrongAttr. Figure 5 shows average latency per request and average number of network messages sent per request for FACADE and Decat *et al.*'s algorithm. When pSameCoord is 0.5 or less, corresponding to deployments with 2 or more coordinators, FACADE has lower latency and sends fewer network messages than Decat *et al.*'s algorithm. FACADE's lower latency stems from using fewer network messages and fewer database queries (due to optimization (3)). Deployments in large systems would probably use around 10 coordinators, as in Decat *et al.*'s experiments. This corresponds to pSameCoord=0.1, for which the average latency of FACADE is *less than half* the average latency of Decat *et al.*'s algorithm (37.7 milliseconds compared to 91.1 milliseconds), and the average network messages per request is 3.8 for FACADE vs. 5.6 for Decat *et al.*'s algorithm. This is true regardless of whether accurate prediction of accessed attributes and written objects is possible. More generally, we see

that incorrect prediction of accessed attributes and written objects have negligible effect on these results. We also see that the average latency of FACADE is almost independent of pSameCoord; this is because local processing time accounts for much of the latency, and the average number of network messages per request changes less for FACADE than Decat *et al.*'s algorithm.

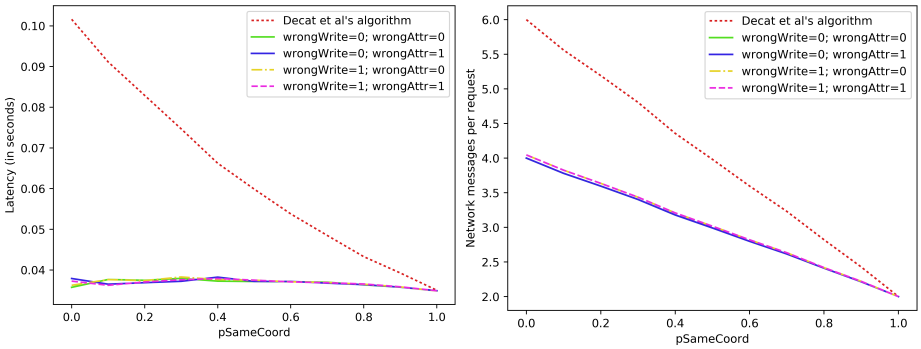

Fig. 5. Average latency per request (left) and average number of network messages per request (right) as a function of pSameCoord.

Throughput. To evaluate throughput, we ran experiments analogous to the performance experiments in [8, section 4.4, Figure 13]. To determine the maximum throughput of each algorithm, we ran experiments with increasing numbers of clients, until the throughput plateaus. For each value of nClient, we ran experiments with increasing numbers of workers, until throughput plateaus. We then used the value of nWorkers determined for the largest value of nClient in experiments with all smaller values of nClient, since we wanted only one workload parameter to vary in the final results. For FACADE with wrongWrite=0 and wrongAttr=0, we found nClient=23 and nWorker=4 provided the maximum throughput of 344 requests/second, with mean latency of 65.5 milliseconds. For Decat *et al.*'s algorithm, we found nClient=19 and nWorker=14 provided the maximum throughput of 318 requests/second, with mean latency of 79.5 milliseconds. Values of the other fixed workload parameters are nObj=1000, nRequest=5000, pWrite=0.1 and pSameCoord=0.1. Figure 6 shows average throughput as a function of nClient for Decat *et al.*'s algorithm and FACADE. For FACADE, average throughput is shown for each of the four possible combinations of values of wrongWrite and wrongAttr. We see that FACADE achieves higher maximum throughput than Decat *et al.*'s algorithm in most cases in these experiments. We also see that FACADE's throughput is more sensitive than its latency to the accuracy of predictions of accessed attributes and written objects.

Local processing time. The CPU time per request for coordinators is similar for FACADE and Decat *et al.*'s algorithm. The CPU time per request for workers is roughly double for FACADE compared to Decat *et al.*'s algorithm, due

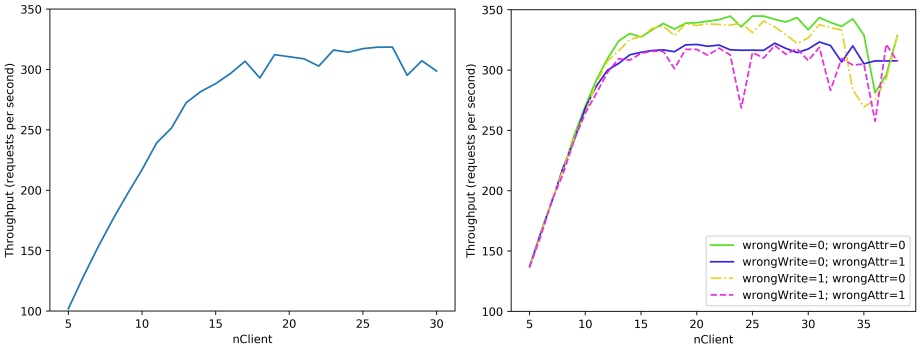

Fig. 6. Average throughput as a function of nClient for Decat *et al.*'s algorithm (left) and FACADE (right).

to versioning and piggybacking data read from the database on messages to coordinators (i.e., optimization (3)). Local processing is a significant fraction of the overall latency (and throughput is relatively low in absolute terms), because Python is relatively slow. If both algorithms were implemented in a faster language such as C++, local processing would be a smaller part of the overall latency, and the ratio of average latency for FACADE to average latency for Decat *et al.*'s algorithm would be even smaller than in our experiments.

Performance with different write probabilities. To evaluate the effect of pWrite on performance, we also ran the latency experiments and throughput experiments (described in the *Latency* and *Throughput* paragraphs above, respectively) with pWrite=0.0 (i.e., all requests are read-only) and pWrite=0.2. We consider pWrite=0.1 to be a realistic value and pWrite=0.2 to be on the high side of the realistic range. pWrite=0.0 is a natural boundary value to consider; it is also the best case for both algorithm's performance. For the latency experiments, the results with pWrite=0.0 and pWrite=0.2 are almost the same as those described above for pWrite=0.1, because writes have little effect on performance when there are no conflicts, and there are no conflicts in experiments with only one client. For the throughput experiment with pWrite=0.0, for FACADE, we found nClient=24 and nWorker=8 provided the maximum throughput of 412 requests/second, with mean latency of 56.6 milliseconds; for Decat *et al.*'s algorithm, we found nClient=24 and nWorker=9 provided the maximum throughput of 373 requests/second, with mean latency of 62.0 milliseconds. For throughput experiment with pWrite=0.2, for FACADE, we found nClient=24 and nWorker=2 provided the maximum throughput of 303 requests/second, with mean latency of 77.4 milliseconds; for Decat *et al.*'s algorithm, we found nClient=25 and nWorker=4 provided the maximum throughput of 283 requests/second, with mean latency of 86.8 milliseconds. Thus, FACADE's maximum throughput is 11%, 8%, and 7% higher than Decat *et al.*'s algorithm's maximum throughput when pWrite=0.0, 0.1, and 0.2, respectively, and FACADE has lower latency in all three experiments.

Performance with more conflicts. To evaluate the effect of a higher conflict rate on performance, we also ran the throughput experiments with an unrealistically small number of objects; decreasing nObj is the simplest way to increase the conflict rate. Specifically, we reduced nObj from 1000 (a more realistic value) to 200 (an unrealistically small value) for these experiments. Other workload parameters, including nClient and nWorker, are the same as described above for the throughput experiments. For FACADE with wrongWrite=0 and wrongAttr=0, the number of restarts due to conflicts increased from 1 to 16, throughput decreased from 344 to 295 requests/second, and average latency increased from 65.5 to 75.9 milliseconds. For Decat *et al.*'s algorithm, the number of restarts due to conflicts increased from 1 to 6, throughput decreased from 318 to 305 requests/second, and average latency increased from 79.5 to 81.7 milliseconds. Although FACADE is more sensitive than Decat *et al.*'s algorithm to this change, FACADE's performance is still competitive, with 3% lower throughput and 7% lower latency than Decat *et al.*'s algorithm.

4 Related Work

Decat *et al.*'s work in [8] is the most closely related and is discussed in previous sections.

Chadwick describes a distributed architecture for a XACML-based stateful policy framework, consisting of multiple policy decision points (PDPs) interacting with a centralized database containing the mutable state [6]. Each PDP locks all relevant rows in the database before evaluating a request. The design has limited scalability, due to the centralized database and locking.

Alzahrani *et al.* describe a similar distributed architecture [1], without committing to a specific approach to storage of the state. They briefly mention a few alternatives, e.g., in a centralized database, or replicated at or partitioned among the PDPs, but do not discuss any of them in detail.

Dhankhar *et al.* consider evaluation of stateful distributed XACML policies. Different PDPs have different policies, and the policies can refer to each other [9]. Concurrency control is provided by a centralized lock manager. Each PDP locks all relevant attributes before evaluating a request. The centralized lock manager limits scalability of their design.

Kelbert and Pretschner describe a fault-tolerant decentralized infrastructure for enforcement of usage control policies [14]. They rely on the database, Cassandra[2], for concurrency control. As mentioned in Section 1, Cassandra provides serializability only for single-row transactions, so their system does not support serializable evaluation of requests involving attributes of two objects.

Weber *et al.* present a framework for stateful access control policies in distributed systems based on weakly consistent replication of the state, as provided by eventually consistent data stores [22]. In contrast, our design is based on the traditional notion of strong consistency. When weak consistency is acceptable, it potentially allows more fault-tolerance and scalability. They do not present a completed implementation or any performance results.

Acknowledgments We thank M. Decat for explaining the details of [8].

References

1. Alzahrani, A., Janicke, H., Abubaker, S.: Decentralized XACML overlay network. In: Proceedings of the 10th IEEE International Conference on Computer and Information Technology (CIT 2010). pp. 1032–1037. IEEE Computer Society (2010)
2. American National Standards Institute (ANSI), International Committee for Information Technology Standards (INCITS): Role-based access control. ANSI INCITS Standard 359-2004 (Feb 2004)
3. Becker, M.Y.: Specification and analysis of dynamic authorisation policies. In: Proc. 22nd IEEE Computer Security Foundations Symposium (CSF). pp. 203–217. IEEE Computer Society (2009)
4. Becker, M.Y., Nanz, S.: A logic for state-modifying authorization policies. ACM Transactions on Information and System Security 13(3) (2010)
5. Brewer, D.F.C., Nash, M.J.: The chinese wall security policy. In: Proceedings of the 1989 IEEE Symposium on Security and Privacy. pp. 206–214. IEEE Computer Society (1989)
6. Chadwick, D.: Coordinated decision making in distributed applications. Information Security Technical Report 12 (2007)
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems 26(2), 4:1–4:26 (2008)
8. Decat, M., Lagaisse, B., Joosen, W.: Scalable and secure concurrent evaluation of history-based access control policies. In: Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015). pp. 281–290. ACM (2015)
9. Dhankhar, V., Kaushik, S., Wijesekera, D., Nerode, A.: Evaluating distributed XACML policies. In: Proceedings of the 4th ACM Workshop On Secure Web Services (SWS 2007). pp. 99–110. ACM (2007)
10. Edjlali, G., Acharya, A., Chaudhary, V.: History-based access control for mobile code. In: Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS '98). pp. 38–48. ACM (1998)
11. Gama, P., Ribeiro, C., Ferreira, P.: A scalable history-based policy engine. In: Proceedings of the 7th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2006). pp. 100–112. IEEE Computer Society (2006)
12. Gay, R., Mantel, H., Sprick, B.: Service automata. In: 8th International Workshop on Formal Aspects of Security and Trust (FAST 2011). Lecture Notes in Computer Science, vol. 7140, pp. 148–163. Springer (2012)
13. ISO/IEC: Information technology — open systems interconnection — security frameworks for open systems: Access control framework. ISO/IEC Standard 10181-3:1996, International Organization for Standardization (2006)
14. Kelbert, F., Pretschner, A.: A fully decentralized data usage control enforcement infrastructure. In: 13th International Conference on Applied Cryptography and Network Security. Lecture Notes in Computer Science, vol. 9092, pp. 409–430. Springer (2015)
15. Liu, Y.A., Stoller, S.D., Lin, B.: From clarity to efficiency for distributed algorithms. ACM Transactions on Programming Languages and Systems 39(3) (2017)

16. Liu, Y.A., Stoller, S.D., Lin, B., Gorbovitski, M.: From clarity to efficiency for distributed algorithms. In: Proceedings of the 2012 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA). pp. 395–410. ACM Press (Oct 2012)
17. Lobo, J., Ma, J., Russo, A., Lupu, E., Calo, S.B., Sloman, M.: Refinement of history-based policies. In: Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 6565, pp. 280–299. Springer (2011)
18. Martinelli, F., Matteucci, I., Mori, P., Saracino, A.: Enforcement of U-XACML history-based usage control policy. In: Proceedings of the 12th International Workshop on Security and Trust Management (STM 2016). Lecture Notes in Computer Science, vol. 9871, pp. 64–81. Springer (2016)
19. Nguyen, D., Park, J., Sandhu, R.S.: A provenance-based access control model for dynamic separation of duties. In: Eleventh Annual International Conference on Privacy, Security and Trust (PST 2013). pp. 247–256. IEEE Computer Society (2013)
20. Park, J., Sandhu, R.: The ucon_{abc} usage control model. ACM Trans. Inf. Syst. Secur. 7(1), 128–174 (Feb 2004)
21. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts. McGraw-Hill, 6th edn. (2011)
22. Weber, M., Bieniusa, A., Poetzsch-Heffter, A.: Access control for weakly consistent replicated information systems. In: Proceedings of the 12th International Workshop on Security and Trust Management (STM 2016). Lecture Notes in Computer Science, vol. 9871, pp. 82–97. Springer (2016)