DIMACS Technical Report 2002-03
January 2002

Regulating E-Commerce through Contract Certificates

by

Victoria Ungureanu[1]
Dept. of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

[1]Visiting Member

ABSTRACT

Enforcing e-commerce contracts is difficult because an enterprise may be concurrently bound by a very large number of commercial agreements, and because these agreements are likely to change in time or to be annulled. We argue that it is not feasible to establish a dedicated server for each contract; nor it is possible to construct a composition of such commercial agreements. To deal with these problems we propose to embed contracts in certificates. We show that disseminating contracts as certificates facilitates deployment, annulment and revision of contracts. We propose a language for stating contract terms, and present several formal examples.

We describe here our implementation, which can be used as an extension to a web server, or as a separate server with interface to application. The proposed model does not require any modification of the current certificate infrastructure, and only minor modifications to servers.

1 Introduction

In order to save money, be competitive and be efficient, more and more enterprises are taking steps towards conducting transactions with trading partners on-line [9]. Among the problems inherent in such projects none is more serious than the difficulty to control the activities of the disparate agents involved in e-commerce.

Trading relations are based on mutually agreed upon contracts. Generally, these contracts enumerate agents authorized to participate in transactions, and spell out such things like rights and obligations of each partner, and terms and conditions of the trade. *Control occurs then as an ancillary to such commercial agreements.*

An enterprise may be concurrently bound by a set of different contracts that regulate the trading relations with its various clients and suppliers. There are currently two main methods to establish a set of policies: (1) to establish a dedicated server for each policy [8, 15, 14], and (2) to *combine* them into a single super-policy [4, 5]. We will argue that both approaches are problematic in e-commerce context.

Having a dedicated server for each contract is an expensive proposition, if the number contracts an enterprise is bound by is very large[1]. This is increasingly the case for many modern enterprises. For example, Ford has approximately thirty thousand suppliers, each operating under a different contract, and General Motors has about forty thousand [10] (both companies have recently announced their intention to perform their inter-enterprise business on-line).

The large number of contracts also makes combination of policies difficult to perform. Moreover, even if it would be possible to create such a composition, it would still be very problematic to maintain it. New contracts are being constantly established, and previously established contracts end, or are being revised. Each such modification triggers, in turn, the modification of the composition-policy, leading to a maintenance nightmare.

Thus, in e-commerce context, a huge, ever-changing policy, subsuming all contracts by which an enterprise is bound, becomes prohibitively hard and error-prone to maintain. And establishing a dedicated server for each contract is simply unpractical. To deal with these problems, we propose, in this paper, to use the certificate framework for contract support. Certificates, by which we mean digitally signed statements of some sort, are commonly used to establish trust between parties who are physical distant or do not know each other. Generally, a certificate conveys information regarding a subject—a software agent or a human user. Certificate based authorization is carried out as follows: An agent submits a request, together with a certificate (or a list of certificates), to a server. The server verifies the credentials and grants or denies the request, according to its *internal, predefined* access control policy.

In e-commerce context, the policy that a server has to enforce is denoted by the agreed contract terms. We are proposing here that: (a) the contract terms, expressed in a formal, interpretable language, should be embedded in certificates, and (b) that an agent making

[1] Roscheisen notes in [14] that in his framework *"the number of objects ends up being quite large in realistic applications, and this quickly becomes infeasible, especially given the resources that each such object requires."*

a request should present to a server a contract-certificate, together with other relevant credentials. The server has to verify that the contract is valid, i.e. it is signed by a trusted issuer, and it has not been revoked. A valid contract-certificate can then be used as the authorization policy for the request in question.

Viewing contracts as certificates makes several aspects of contract management easier:

- `deployment:` Contracts sanctioned by an enterprise may be stored on repositories (such as a Web server), from where agents retrieve contract-certificates as needed. New contracts are established simply by embedding them in certificates and publishing them on designated repositories.
- `annulment:` Contract annulment can be modeled conveniently by certificate revocation.
- `revision:` In practice contracts may be revised for various reasons. If a contract needs to be revised, this can be done simply by publishing a new contract-certificate, and by revoking the old one.

The rest of the paper is organized as follows: We start, in Section 2, by describing how contracts can be modeled as certificates, and by illustrating this concept with an example. We follow, in Sections 3 and 4 by discussing how contract-certificates can be deployed, revoked and updated. Section 5 presents an extension of the concept, which allows expressing contract provisions terms as a function of *dynamically modifiable* state. The system implementation is introduced in Section 6. Section 7 discusses related work, and we conclude in Section 8.

2 Expressing E-commerce Contracts as Certificates

We have argued that, so far, certificates contained information regarding subject identity, like name, public key, or group membership. We are interested here in extending the certificate concept, so that it may convey not only subject-credentials per-se, but also information regarding the contract(s) a subject is bound by. This type of certificate should mirror as closely as possible the social notion of contract, while tailoring it for the specific needs of e-commerce.

At the societal level, a contract embodies an agreement between two or more parties involved in a certain (economic) activity and refer to such things as: the *time frame* in which the activity in question is to be completed, the *agents* authorized to participate, the expected *rules* of conduct of participating agents, and the *penalties* incurred for not complying with the rules. It follows that a contract-certificate should define, in a formal language, these types of contract terms. One possible language for expressing contract rules, and a concrete example of rules implementation will be presented in the following paragraph.

In practice, contracts may be nullified before the end of their validity period due to changes of legislation, bankruptcy, etc. Contract annulment can be modeled conveniently by certificate revocation. We assume that each contract-certificate contains a reference to a revocation server, which is trusted with maintaining and propagating information regarding contract annulment.

Contract annulment may occur when there is a radical and sudden change in the legal context, or in the business conditions of either party. Often though, such changes are moderate and can be handled by contract revisions, without resorting to contract nullification. In order to support contract updates, we assume that each contract is identified by its name and a version number; and the latest version of a contract is maintained by a repository, whose address is given in the contract-certificate.

Traditional certificates are presented by the bearer to prove identity or group membership; similarly, contract-certificates are presented to show compliance with an a-priori set-up contract. The underlying assumption, in both cases, is that there is a server which is trusted to verify credentials, and to grant access accordingly. In order to ease processing, contract-certificates contain a special attribute, `type(contract)` which enable servers to distinguish between contract and identity certificates. And to ease the server-location process, a contract-certificate may have an attribute `servedBy` whose value denotes the address of a server, trusted by the participating parties with enforcing contract terms.

To summarize, the following are the mandatory components of a contract-certificate:

- `type`—denotes the type of a certificate. It is the presence of the term `type(contract)` which enables a server to distinguish between subject and contract certificates.
- `name`—denotes the id of the contract represented by the certificate;
- `version`—denotes the version of the contract carried by the certificate. The version number, as we shall see, provides support for contract revisions;
- `validity period`—denotes the validity period of the contract;
- `revocationServer`—specifies the address of the server maintaining/disseminating information regarding contract revocation.
- `repository`—specifies the address of the server which maintains information about the contract in question. Namely, we assume that a `repository` maintains the latest version of a contract.
- `contract terms`—specify, in a formal language, the contract terms.

A contract-certificate is obtained by signing a statement comprised by these, and possibly other, attributes by an issuer trusted by participating parties.

Expressing Contract Terms Contract terms can be quite naturally expressed by means of any formal language supporting *event-condition-action* (ECA) kind of rules. We are using here an extension of a language devised for support of control policies [12, 13] built on top of Prolog [6]. But the nature of this language is, in a sense, of a secondary importance.

Operations on certificates	
`issuer(pk,cert)`	binds `pk` to the public key of the issuer of certificate, `cert`;
`role(r,cert)`	binds `r` to the value of attribute `role` in `cert`, if one exists; fails otherwise;
`name(n,cert)`	binds `n` to the value of attribute `name` in `cert`, if one exists; fails otherwise;
`bind(attr,val,cert)`	binds `val` to the value of attribute `attr` in `cert`, if one exists; fails otherwise;

Miscellaneous	
`accept`	denotes that the request is valid;
`reject`	denotes that the request is invalid;
`deliver(m,d)`	delivers message `m` to destination `d`.

Figure 1: Some Primitive Operations

For now, contract terms are defined by means of Prolog-like rules which, evaluate agent requests, and decide whether they are valid or not, taking into account the request and the credentials presented. More specifically, a rule for a request `R`, accompanied by certificate(s) `Cert` of the form:

```
eval(R,Cert):-
condition-1,...,condition-k,
provision-1,...,provision-n,
do(accept).
```

states that, if *condition-1* through *condition-k* are satisfied then the request is valid. These conditions may refer to the content of the request, and of the certificates presented. Moreover, the rule calls for carrying out *provision-1* through *provision-k*, which denote any additional actions required by the contract.

In addition to the standard types of Prolog goals, the body of a rule may contain a distinguished type of goal, called a *do-goal*. A do-goal has the form `do(p)`, where `p` is a *primitive-operation*. A sample of primitive operations is presented in Figure 1.

An Example We demonstrate here how contracts can be expressed into contract-certificates (abbreviated as `CC`), by presenting informally a simple contract, and showing how it can be represented by a `CC`.

Agents in a client enterprise, say Ford, may purchase audio equipment from a supplier enterprise, say RCA, provided that:

- *purchase offers are be issued between November 1 and December 1, 2001.*

- *only agents duly certified as purchase officers by* `Ford-CA`[2]*, a designated certification authority of the client enterprise, may issue purchase orders.*
- *Only agents duly certified as sale representatives by* `RCA-CA`*, a designated certifying authority of the supplier enterprise, are authorized to respond to POs.*
- *A copy of all accepted POs must be sent to a designated audit-trail.*

This contract is formalized by the contract-certificate displayed in Figure 2. This figure has two parts, specifying the *preamble* to the contract, and its rules. Each rule is followed by a comment, in italic, which, together with the following discussion should be understandable even for a reader not well versed in Prolog.

The preamble to this contract has several clauses specifying: the name of the contract (`ford-rca`, in this case), its validity period, the addresses of the revocation server and of the repository. Our discussion of the contract rules is organized as follows: We start with how a client-agent may issue a purchase order by presenting a specified type of certificate, signed by the stipulated authority. We will then show how a vendor-agent may respond to such a purchase order[3].

By Rule $\mathcal{R}1$, a PO is considered valid only if the sending agent presents a certificate, `ClientCert`, issued by `clientAuthority` which certifies the bearer to be a `purchaseOfficer`.

Under this contract, a vendor agent may respond to a purchase order with a `responseToOffer` message, which can contain a positive result (an accept), or a decline of the offer. Such a message is considered valid if the vendor has presented a certificate issued by `supplierAuthority`, certifying it to be a `saleRepresentative` (Rule $\mathcal{R}2$). Moreover, if the offer is accepted, a copy of the PO is sent to the designated audit-trail. Finally, by Rule $\mathcal{R}3$, all other requests are rejected as being invalid under this contract.

3 Contract-Certificate Deployment and Revocation

Deployment Two components are necessary for deploying contract-certificates: repositories—to maintain and disseminate contract-certificates; and severs—to interpret contract terms and bring them to bear. We start by describing their functionality, and follow with a brief discussion about their deployment.

Contracts can be established between two enterprises (business-to-business, or B2B, commerce) or between an enterprise and an individual client (business-to-customer, or B2C, commerce). In either case, an enterprise can code the contracts it is abiding to as certificates, and publish them on repositories from where they can be fetched as needed. Repositories, are an an integral part of certification infrastructure, and can be used for contract-certificates without any modification.

[2] In this example, for sake of convenience, `Ford-CA` stands for the public key of the certifying authority trusted to sign purchase officer certificates on behalf of the client enterprise.

[3] This is only a finger exercise, meant to illustrate the concept of a contract-certificate; a full contract should consider other types of requests, including delivery notices, acknowledgments, etc.

Preamble:

type(contract).
name(ford-rca).
version(1).
validity([1,nov,2001],[1,dec,2001]).
repository(http://trust.ibm.com/).
revocationServer(http://trust.ibm.com/)).

$\mathcal{R}1$.
```
eval(purchaseOffer(Specs,Amount,PaymentInfo,Vendor),ClientCert) :-
        do(issuer(clientAuthority,ClientCert)),
        do(role(purchaseOfficer,ClientCert)).
        do(accept).
```

A `purchaseOffer` *request is authorized if the sender is established as a* `purchaseOfficer` *by a certificate issued by* `clientAuthority`*.*

$\mathcal{R}2$.
```
eval(responseToOffer(Response,Specs,Client),VendorCert) :-
        issuer(vendorAuthority,VendorCert),
        role(saleRepresentative,VendorCert), if Response==accept then
        do(deliver(Specs),auditor),
        do(accept).
```

A `responseToOffer` *message is considered valid if the sender is an agent established as a* `saleRepresentative` *by* `vendorAuthority`*. Moreover, if the PO is accepted a copy is sent to the* `auditor`*.*

$\mathcal{R}3$.
```
eval(M,Cert) :- do(reject).
```

All other requests are rejected.

Figure 2: A contract-certificate example.

This is not true, however, for servers which need to be modified in order to deal with contract-certificates. We have developed a generic contract-engine, called observer, that can interpret any well-formed contract. We are assuming that each server processing e-commerce requests has one associated observer, to which it passes received requests for evaluation. Servers are trusted to service only requests sanctioned by observers. As an implementation example, we will show in Section 6 how we have extended a Web server to use observers.

Such servers may be maintained by organizations trusted by all parties involved in a contract, like large, trustworthy, financial institutions—such as Visa or Master Card, service providers—like Ebay, or software providers—such as Sun or Microsoft. If there is no such trusted intermediary, any enterprise interested in enforcing the terms of an agreement can establish its own servers. In many cases, this is a workable proposition, because an enterprise values too much its reputation to break contract terms purposely, by installing rogue servers. We mention that, since contract-certificates are signed, repositories may be maintained by any party involved or by a third party without creating security breaches.

Revocation Certificates might become invalid for various reasons and should be revoked. (For example, the secret key authenticated by a certificate might be lost or be compromised, or the owner information, like role or address, might change.) Most revocation mechanisms rely on the existence of *certificate revocation lists* (CRL) maintained by trusted revocation servers ([2, 16]). To revoke a certificate, the owner or another responsible authority, sends the revocation server a signed message identifying the certificate to be revoked. Upon receipt of the message, the revocation server updates its CRL and disseminate the information [18].

Depending on the method used for disseminating revocation information one distinguishes between pull and push-based systems. In pull based systems, applications, who want to check the validity of a certificate, query the revocation server, and in response receive all or part of the latest, signed CRL. In push based systems, signed CRL updates are sent out periodically to interested parties.

Similarly, contracts may be annulled before their expiration date, because, for example, one of the parties is bankrupt. Like traditional certificates, a `CC` may be revoked by its issuer by sending a message to the revocation server. It is worth noting that both push or pull based methods may be used.

4 Contract Update

Generally contracts are not immutable, and in practice, revisions of a contract may be called for various reasons, like, for example, to correct omissions, to better accommodate the needs of either party, or to reflect unplanned circumstances. To be more specific, consider again our contract-example. Here are some possible scenarios that would require its modification:

1. *the group of agents allowed to issue a purchase order is extended to also include duly certified managers.*

2. *only duly certified managers may issue purchase offers in excess of a certain amount, say $1000.*

3. *the vendor enterprise establishes a new certifying authority,* `vendorAuthority'` *which can issue certificates enabling agents to respond to purchase offer.*

In traditional control mechanisms, such changes take effect by manually implementing a revised access control policy into *each* server that uses the policy in question—a laborious and error-prone process. If however, contracts are implemented as certificates, contract revision can be handled simply by: (a) issuing a new version of the contract whose rule reflect the updates deemed necessary, and (b) revoking the previous, obsolete version.

To show how contract-update can be supported, we present now a new version our contract-example that incorporates all the revisions mentioned above. This version of the contract is shown, in its entirety, in Figure 3.

The preamble of this contract specifies that the second version of `ford-rca` contract is valid from 11/15/2001 until 12/01/2001. The revisions are materialized into contract rules as follows. Point 1 is brought to bear by Rule $\mathcal{R}1$ which states that agents duly certified as `manager` by `clientAuthority` may issue purchase orders regardless of the amount of the order. However, by Rule $\mathcal{R}2$, of this contract version, a PO issued by a duly certified `purchaseOfficer` is accepted only if the amount offered is less then $1,000, thus realizing point 2. Finally, point 3, is implemented by Rule $\mathcal{R}3$, stating that an agent may respond to a purchase order if it is certified as a `saleRepresentative` by either `vendorAuthority` or `vendorAuthority'`.

Dealing with Outdated Requests We are dealing now, with the case where a request `r` has been received under a version of the contract, which has expired *before* `r` could be evaluated. This situation may occur if, for example, the server has been back-logged with requests; or the server has incurred delays verifying credentials.

To handle these case we introduce into the language, a new type of rule, namely

```
arbiter(VersionNo, Request, Credentials),
```

which takes three parameters: the version of the contract under which this request was supposed to be handled (now, invalid); the request itself; and the credentials accompanying the request in question. An `arbiter` rule specifies how `Request` should be treated. Here are some possibilities:

- provide for a "grandfather clause", namely, the request should be validated provided it mets the conditions specified by the contract-version under which it was received;

- provide for some special treatment;

- reject the request, and notify the interested parties of the outcome.

$\mathcal{P}$*reamble:* type(contract).
name(ford-rca)).
version(2).
validity([15,nov,2001],[1,dec,2001]).
repository(http://trust.ibm.com/).
revocationServer(http://trust.ibm.com/).

$\mathcal{R}1$.
```
eval(purchaseOffer(Specs,Amount,PaymentInfo,Vendor),ClientCert) :-
        issuer(clientAuthority,ClientCert),
        role(manager,ClientCert),
        do(accept).
```

A `purchaseOffer` *request issued by an agent certified as a* `manager` *by* `clientAuthority` *is authorized regardless of the amount of the order.*

$\mathcal{R}2$.
```
eval(purchaseOffer(Specs,Amount,PaymentInfo,Vendor),ClientCert) :-
        issuer(clientAuthority,ClientCert),
        role(purchaseOfficer,ClientCert),
        Amount <=1000, do(accept).
```

A `purchaseOffer` *request is authorized if the following conditions are met: (1) the sender is established as a* `purchaseOfficer` *by a certificate issued by* `clientAuthority` *and (2) the* `Amount` *the agent is willing to pay for the merchandise is less then $1000.*

$\mathcal{R}3$.
```
eval(responseToOffer(Response,Specs,Client),VendorCert) :-
        (issuer(vendorAuthority,VendorCert) or
         issuer(vendorAuthority,VendorCert)),
        role(saleRepresentative,VendorCert),
        if Response==accept then do(deliver(Specs),auditor),
        do(accept).
```

A `responseToOffer` *message may be issued only by an agent certified as a* `saleRepresentative` *by either* `vendorAuthority` *or* `vendorAuthority'`.

$\mathcal{R}4$.
```
eval(M,Cert) :- do(reject).
```

All other requests are rejected.

Figure 3: Revised version of the contract-example.

To illustrate the use of "arbiter"-rule consider again the contract-example. Under the first version of the contract, a `purchaseOffer` was accepted, regardless of the amount offered, provided only it was issued by a duly authorized `purchaseOfficer`. Under its revision, however, a `purchaseOfficer`, may issue POs only if the amount offered does not exceed $1,000. The question is: how should a purchase order arriving while the first version was valid, but processed when after it has been revoked revoked, be treated? In this case we chose to resolve the rule conflict by providing for a grandfather clause (see Rule $\mathcal{R}5$ in Figure 4).

$\mathcal{R}5$.
```
arbiter(version(1),purchaseOffer(Specs,Amount,PaymentInfo,Vendor),
        ClientCert) :-
        issuer(clientAuthority,ClientCert),
        role(purchaseOfficer,ClientCert),
        do(accept).
```

A `purchaseOffer` *request processed under version 2 of the contract is considered valid if: (1) it was received under the first version of the contract, and (2) it mets the conditions stated in that contract version, namely, it was sent by a duly authorized* `purchaseOfficer`*.*

Figure 4: Revised version of the contract-example (cont.)

5 Extending the Expressive Power of Contract-Certificates

So far, contract terms have been formulated by taking into account the request content, and any credentials the requester might present. While this is sufficient for many types of commercial agreements, there are cases when the terms of a contract take also into account additional information—referred here as control state. The control state, which is likely to *change dynamically* during the lifetime of a contract, might consist, for example, of: the different phases of an agreement, the state of transactions regulated by the contract, or the past actions of various agents.

To illustrate the range of contracts we have in mind, we introduce into the following amendments to our contract-example introduced in Section 2.

- *RCA, the supplier enterprise, honors purchase orders (POs) issued by Ford, for up to a cumulative value—to be called "blanket"— of $10,000.*
- *a purchase offer should be accepted or declined in no more than one day— after this time limit, the offer is considered void.*

This contract, which we call the blanket-contract, is stateful since it is sensitive to the value of the blanket, and to the times a PO, and a response to a PO, are made.

In order to support this type of contracts, we are extending the language with the following constructs. First, a `homeServer` clause, which contains a reference to a server address.

Operations on the control-state	
`t@CS`	returns true if term `t` is present in the control state, and fails otherwise
`+t`	adds term `t` to the control state;
`-t`	removes term `t` from the control state;
`incr(t(v),d)`	increments the value of the parameter `v` of term `t` with quantity `d`
`dcr(t(v),d)`	decrements the value of the parameter `v` of term `t` with quantity `d`
Miscellaneous	
`time(t,u)`	binds `t` to the current time, expressed using units `u` ;

Figure 5: Some Primitive Operations

This server, mutually agreed by interested parties, is trusted to maintain correctly the control state during the life time of the contract. Second, an `initialCS` clause which specifies the initial value of the control state. For example, the formalization of the contract above, should contain the clause `initialCS[blanket(10,000)])`. It is the presence of this type of clause that enables a server to distinguish between a stateful and a stateless contract. Finally, the language for expressing contract terms should provide for additional primitive operations that test the value of the control state and allow for its modifications. These primitives, and an additional primitive needed to implement the example, are presented in Figure 5. The formalization of the blanket contract is displayed in Figure 6.

Discussion It is worth noting here that control state support comes with a price: a certificate, denoting a stateful contract, will be honored *only* by the `homeServer` mentioned in the certificate. This is a severe limitation on the intended power of certificates, which are meant to be used anywhere. This restriction, however, affects only stateful contracts; contracts whose terms can be expressed without resorting to state can be embedded in certificates which retain their full power.

Control State Update Like contract terms, the state may need to be revised for various reasons. For example, consider again the blanket contract, and suppose that the volume of purchases between enterprises is larger than expected, and that the initial value of the blanket proved to be insufficient. Suppose further, that, as a consequence, both enterprises agree to increase the blanket value to $15,000 (while the rest of the provisions remain unchanged). Implementing such a revision requires updating the value of the control state.

To handle state updates effectively we introduce into the language a new type of rule

```
update(from(N)),
```

where `N` is the number of the previous contract version. This type rule that mandate if, and, how the control state should be modified. And is triggered if the server, which is currently maintaining version `N` of a given contract `C`, detects that `C` has been revised (i.e. that the

Preamble: type(contract).
name(ford-rca).
version(3).
validity([20,nov,2001],[1,dec,2001]).
repository(http://trust.ibm.com/)).
revocationServer(http://trust.ibm.com/)).
homeServer(sentry.ibm.com).
initialCS([blanket(10000)].

*R*1.
```
eval(purchaseOffer(Specs,Amount,PaymentInfo,Vendor),ClientCert) :-
        do(issuer(clientAuthority,ClientCert)),
        do(role(purchaseOfficer,ClientCert)),
        blanket(B)@CS, B>Amount, do(dcr(blanket(B),Amount)),
        do(time(T,h)), do(+order(Specs,Amount,T)),
        do(accept).
```

A `purchaseOffer` *request is authorized if the following conditions are met: (1) the sender is established as a* `purchaseOfficer` *by a certificate issued by* `clientAuthority`*, and (2) the value of the blanket exceeds the* `Amount` *the client is willing to pay. If this is the case, the blanket value is decreased accordingly. Moreover, a term* `order(Specs,T)` *is added to the control state, to record the time the purchase order for* `Specs` *was made.*

*R*2.
```
eval(responseToOffer(accept,Specs,Client),VendorCert) :-
        issuer(vendorAuthority,VendorCert),
        role(saleRepresentative,VendorCert),
        order(Specs,Amount,T1)@CS, do(time(T2,h)), T2-T1<24,
        do(deliver(Specs,auditor)),
        do(accept).
```

If the vendor accepts the offer, the response is considered valid if the following conditions are met: (1) the sender is an agent established as a `saleRepresentative` *by* `vendorAuthority`*; and (2) the response came no later than 24h.*

*R*3.
```
eval(responseToOffer(decline,Specs,Client),VendorCert) :-
        issuer(vendorAuthority,VendorCert),
        role(saleRepresentative,VendorCert),
        order(Specs,Amount,T1)@CS,blanket(B)@CS,
        do(incr(blanket(B),Amount)),
        do(accept).
```

However, if the vendor declines the offer, the blanket is increased by the `Amount` *the client was willing to pay for the merchandise. Note, that, as before, only an authorized* `saleRepresentative` *may send a decline notice.*

*R*4.
```
eval(M,Cert) :- do(reject).
```

All other requests are rejected.

Figure 6: An example of a stateful contract: the blanket contract.

$\mathcal{R}1$. `update(version(3)) :- blanket(B)@CS, do(incr(blanket(B),5000)).`

When a change in version is detected, the control state is updated before any request is evaluated. In this example, the value of the budget term, established by the version 3 of the contract, is increased by 5,000.

Figure 7: Fragment of a revision of the blanket contract handling state updates.

version has changed). Note that `update` rules are triggered *before* any request arriving under the new version, say `N'`, is evaluated.

6 Implementation

The system architecture, illustrated in Figure 8, relies on the existence of three trusted entities: revocation servers, contract repositories and contract engines. As we have mentioned contract engines, called observers, are generic tools that can verify certificates, maintain relevant information regarding contracts they serve, and interpret and carry out contract terms. Additionally, each such observer maintains a list of trusted contract issuers, `TI`. Under this scheme, an application server (e.g. Web, database, or e-mail server) has an associated observer, to which it passes received requests for evaluation. Servers are trusted to service only requests sanctioned by observers.

We are in position now to explain how contract enforcement is carried out. Consider that a user `U` makes a request `R`, to a server `S`, which has an associated observer `O`. Assume further that `R` is accompanied by a contract certificate `CC`, and by a list (possibly empty) of subject-certificates. Then the following steps will be taken:

- The server, `S` passes the request, and the certificates to the observer `O` to decide whether the request should be served or not.
- `O` verifies that `S` is authorized to serve requests issued under the contract designated by `CC`. This implies checking if: (1) `CC` has been signed by a principal belonging to the list of trusted issuers, `TI`; and (2) if the contract specifies a home server (by means of a `homeServer` clause), then `S` matches the specification.
- `O` checks if request `R` has arrived during the validity period inscribed in the certificate.
- `O` checks if the contract is still valid. Assuming that a pull scheme is used, this implies contacting the revocation server mentioned in the certificate, and retrieving the latest CRLs.
- If the contract is valid, `O` records the request, and the contract-version under which it was received. This is done as follows: the observer retrieves `CN`, the contract name, and `VN`, the contract version number from `CC`, the contract-certificate. If there is already an entry for `CN`, the observer records into it the pair `(R, VN)` . Alternatively, if this is

the first request to be processed under `CN`, then `O` sets up an entry for it, in which it records the current version, the rules of the contract, and the pair (`R`, `VN`). Additionally, if the contract is stateful, `O` initializes the control state (with the value indicated by the `initialCS` clause) and records it.

- In case the contract has been revoked, `O` searches for an update of this contract. For this, it asks the repository mentioned by `CC` for the latest version of `CN`. Assuming that the repository maintains a certificate `CC'`, containing version `VN'` of the contract, then `O` takes the following steps: First it checks whether, `CC'` itself has not been revoked. Second, it updates the contract-version number and the contract rules, and records the pair (`R`, `VN'`). Finally, `O` updates the control state of the contract, by evaluating the `update` rules, if any, found in the contract revision.

- If the request is accompanied by user-certificates then, for each such certificate `UC`, an attempt is made to confirm that `UC` is a valid certificate. That implies verifying that: (1) the signature is correct, (2) that the certificate belongs to `U`, and (3) the certificate has not been revoked.

- Finally, `O` checks whether there is a rule in the contract authorizing the request and carries out the ruling. This is done as follows. First, the observer tries the `eval` rules. If none is found, then, if appropriate, it checks the `arbiter` rules. (An `arbiter` rule may be triggered only if there is a discrepancy between the current version of the contract, and the version number recorded together with the request.) If there is a rule sanctioning the request, then the observer: (1) computes and records the new state, in case `CN` is stateful, and (2) carry out any additional provisions. In the end, the server is informed of the outcome; if the request was authorized then, `S` processes the request; otherwise, the request is discarded.

In the current implementation we are using the Jigsaw server [7], developed by W3 Consortium, which has been modified to communicate with observers. Observers are implemented mostly in Java, and operate as independent processes. An observer communicates by pipes with the server it is associated with. To ensure state consistency a controller evaluates the rules pertaining to the same contract *sequentially*, and carries them out *atomically*, so that the sequence of operations that constitutes the ruling for one request does not interleave with those of any other request sent under the same contract.

7 Related Work

There has been a growing interest in supporting e-commerce contracts, and a variety of different, and quite powerful, enforcement mechanisms have been devised, like for example [8, 15, 1, 12]. However, to the best of our knowledge, none of the frameworks proposed so far, express contracts by means of certificates, nor do they deal with contract annulment or

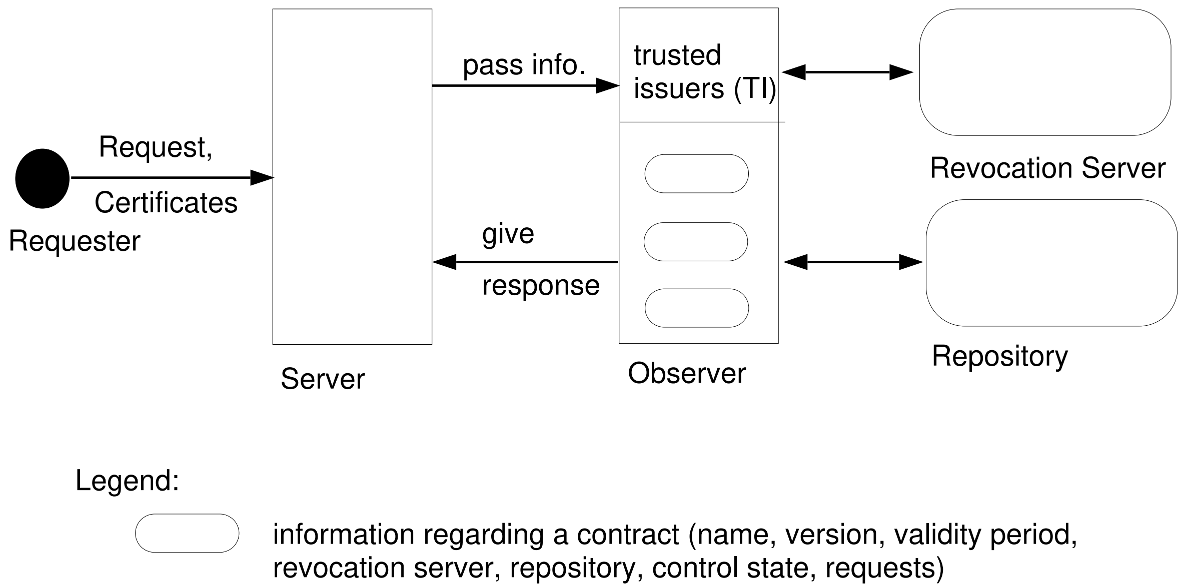

Figure 8: System architecture

revision. This is not surprising, given the difficulty of supporting policy update in traditional frameworks.

There are few researchers which, like us, propose expressing various types of control policies as certificates. Theimer et al. [17] and Aura [3] proposed that delegation policies should be distributed as certificates; and Ioannidis et al. [11] advocated the use of certificates for dissemination of network security policies. The main difference between these works and ours boils down to the disparate views we have on the content of these certificates. In their view, a certificate contains *both* the identity of a user (given as his public key) and the policy fragment spelling out his rights. In this case, a policy is established by issuing certificates for *all* agents having privileges under the policy in question. And if a policy needs to be revoked, or revised, the certificates of *all* agents, which have rights under the policy in question, have to be updated. This is difficult to achieve, especially if the number of policies and/or the number of agents is large.

In our view, we have two distinct types of certificates: subject-certificates, which establish the identity of their owners, and contract-certificates which describe the rights of all agents operating under the contract. In this case, to establish a contract, a *single* certificate has to be issued. And if the contract is annulled or revised, all it is needed is to revoke/update the certificate embedding the contract. Of course, this approach is practical only in the case the contract terms refer to groups of agents, or to the roles that they might take.

8 Conclusion

The well known principle of separation between policy and mechanism is carried out in traditional access control mechanisms by having servers enforce any policy expressible in a certain model. However, in most implementations, a server is associated with a *single* policy—which has been loaded a-priori. In the framework presented here, a server is no longer bound by a unique policy: contracts, embedded in certificates, are presented by requesters in an ad-hoc manner, as the need arise. This enhanced flexibility has several important benefits, especially in the B2B e-commerce context. First, one does not have to maintain a dedicated server for each contract (or set of contacts). As such, the number of agreements, in which an enterprise is involved in, is no longer an issue. Second, it is easy to establish new contracts: all that is required, is to embed them in certificates, and deploy them on repositories. Finally, contract revision and annulment can be supported with great ease by using existing certificate infrastructure.

References

[1] S. Abiteboul, V. Vianu, B. Forham, and Y. Yesha. Relational transducers for electronic commerce. In *Symposium on Principles of Database Systems*, pages 179–187, June 1998.

[2] C. Adams and R. Zuccherato. Internet x.509 public key infrastructure data certification server protocols. Technical report, Internet Draft, PKIX Working Group, 1998.

[3] T. Aura. Distributed access-rights management with delegations certificates. In *Secure Internet Programming*, pages 211–235, 1999.

[4] C. Bidan and V. Issarny. Dealing with multi-policy security in large open distributed systems. In *Proceedings of 5th European Symposium on Research in Computer Security*, pages 51–66, September 1998.

[5] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proc. of the Seventh ACM Conference on Computer and Communications Security*, pages 164 – 173, Athens, Greece, 2000.

[6] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[7] World Wide Web Consortium. Jigsaw - the W3C's web server. website:http://www.w3.org/Jigsaw/.

[8] A. Dan, D.M¿ Dias, R. Kearny, T.C Lau, T. N. Nguyen, F. N. Parr, M. W. Sachs, and H. H. Shaickh. Business-to-business integration with tpaML asnd a business-to-business protocol framework. *IBM Systems Journal*, 40(1):68–90, 2001.

[9] Economist. E-commerce (a survey). pages 6–54. (The February 26th 2000 issue).

[10] Economist. Riding the storm. pages 63–64. (November 6th 1999 issue).

[11] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[12] N.H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *The 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 322–331, Amsterdam, The Netherlands, May 1998.

[13] N.H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000.

[14] M. Roscheisen. *A Network-Centric Design for Relationship Based Rights Management.* PhD thesis, Stanford University, 1997.

[15] M. Roscheisen and T. Winograd. A communication agreement framework for access/action control. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.

[16] International Telegraph and Telephone Consultative Committee. The directory-authentification framework, recommendation. 1993 update.

[17] M. Theimer, D. Nichols, and Douglas Terry. Delegation through access control programs. In *Proceedings of Distributed Computing System*, pages 529–536, 1992.

[18] R. Wright, P. Lincoln, and J. Millen. Efficient fault-tolerant certificate revocation. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, November 2000.