

Regulating E-Commerce through Certified Contracts

Victoria Ungureanu*
Rutgers University
180 University Ave.
Newark, NJ, USA
ungurean@rbs.rutgers.edu

Abstract

Access control has traditionally assumed a single, monolithic authorization policy, generally expressed as an access matrix. We argue that this assumption does not fit e-commerce applications, which are governed by a potentially large set of independently stated, evolving contracts. In order to support this growing class of applications we propose an enforcement mechanism which uses certified-contracts as authorization policies. A certified-contract is obtained: (a) by expressing contract terms in a formal, interpretable language, and (b) by having it digitally signed by a trusted principal. We show that this approach would make dissemination, revision, and annulment of contracts more manageable and more efficient.

We propose a language for stating contract terms, and present several formal examples of certified contracts. The paper describes the implementation of the enforcement mechanism, 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 [8]. 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 (for e.g. [7, 14]), and (2) to combine them into a single super-policy (for e.g. [3, 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 of contracts an enterprise is bound by is very large. 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 [9] (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 fol-

*Work supported in part by DIMACS under contract STC-91-19999 ITECC, and by Information Technology and Electronic Commerce Clinic, Rutgers University

lows: 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 digital signatures should be used not only to certify the credentials a user presents, but also to authenticate the contract rules that a server uses for authorization purposes. Namely, this approach assumes that: (a) the contract terms expressed in a formal, interpretable language are digitally signed by a proper authority, and (b) an agent making a request presents to a server such a certified-contract (abbreviated here as CC), together with other relevant credentials. A valid certified-contract can then be used as the authorization policy for the request in question.

It is the thesis of this paper that this approach would make several aspects of contract enforcement more manageable and more efficient:

- **deployment** : Contracts sanctioned by an enterprise may be stored on repositories (such as a Web server), from where agents may retrieve certified-contracts as needed.
- **annulment** : Contract annulment can be modeled conveniently by certificate revocation.
- **revision** : If a contract needs to be revised, this can be done simply by publishing a new certified-contract, and by revoking the old one.

It is worth noting that all of the above operations can be performed in a *scalable* manner, since their performance does not depend on the number of servers that enforce a given contract, nor on the number of contracts that are supported by a given server.

The rest of the paper is organized as follows: We start, in Section 2, by describing how contracts can be modeled as CCs, and by illustrating this concept with an example. We follow, in Sections 3, 4 and 5 by discussing how certified-contracts can be deployed, revoked and updated. 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 Certified Contracts

When formalizing contracts into CCs our goal is to 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 certified-contract 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 Section 2.1.

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 provided that there are trusted **revocation servers** which maintain and propagate information regarding contract annulment, and that each contract contains a reference to the appropriate revocation server.

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 a **name** and a **version number**; and that the latest version of a contract is maintained by a **repository**, whose address is given in the certified-contract.

Traditional certificates are presented by the bearer to prove identity or group membership; similarly, certified contracts 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, certified-contracts should contain a special attribute, `type(contract)`, which enable servers to distinguish between CCs and traditional, subject certificates. And to ease the server-location process, a certified-contract may have an attribute `servedBy` whose value denotes the address of a server, trusted by the participating parties with enforcing contract terms.

To summarize, we propose that a certified contract should contain the following mandatory components:

- **type** — denotes the type of a certificate. It is the presence of the term `type(contract)` which enables a server to distinguish between subject certificates and CCs.
- **name** — denotes the id of the contract represented by the CC;
- **version** — denotes the version of the contract carried by the CC. 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.

Finally, a certified-contract is obtained by signing a statement comprised of these, and possibly other attributes by an issuer trusted by all participating parties.

2.1. 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. But the nature of this language is, in a sense, of a secondary importance.

In this language contract terms are embedded in rules of the form:

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

This rule 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*, denoting 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 certified-contracts, by presenting informally a simple contract, and showing how it can be represented by a CC. As a simple example of a contract, consider that agents in a client enterprise, say Ford, may purchase audio equipment from a supplier enterprise, say RCA, provided that:

- *purchase offers are to be issued between June 1 and July 1, 2002.*
- *only agents duly certified as purchase officers by `ford_CA`¹, a designated certification authority of the client enterprise, may issue purchase orders (POs).*
- *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 certified-contract 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, the version number, 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 order². First, by Rule $\mathcal{R}1$, a PO is considered valid only if the sending agent presents a certificate, `ClientCert`, issued by `ford_CA`, which certifies the bearer to be a `purchaseOfficer`.

Second, 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 only if the vendor has presented a certificate issued by `rca_CA`, 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. Deployment of Certified Contracts

Two components are necessary for deploying certified-contracts: repositories—to maintain and disseminate certified-contracts; and servers—to interpret contract terms

¹In this example, for the 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.

²This is only a finger exercise, meant to illustrate the concept of a certified-contract; a full contract should consider other types of requests, including delivery notices, acknowledgments, etc.

Operations on certificates	
<code>issuer(pk, cert)</code>	binds <code>pk</code> to the public key of the issuer of certificate <code>cert</code> ;
<code>role(r, cert)</code>	binds <code>r</code> to the value of attribute <code>role</code> in <code>cert</code> , if one exists; fails otherwise;
<code>name(n, cert)</code>	binds <code>n</code> to the value of attribute <code>name</code> in <code>cert</code> , if one exists; fails otherwise;
<code>bind(attr, val, cert)</code>	binds <code>val</code> to the value of attribute <code>attr</code> in <code>cert</code> , if one exists; fails otherwise;
Miscellaneous	
<code>accept</code>	denotes that the request is valid;
<code>reject</code>	denotes that the request is invalid;
<code>deliver(m, d)</code>	delivers message <code>m</code> to destination <code>d</code> .

Figure 1. Some primitive operations

```
Preamble:
type(contract).
name(ford-rca).
version(1).
validity([1,june,2002],[1,july,2002]).
repository(http://trust.intertrust.com/).
revocationServer(http://trust.intertrust.com/)).

R1. eval(purchaseOffer(Specs, Amount,
    PaymentInfo, Vendor), ClientCert) :-
    do(issuer(ford_CA, 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 ford_CA.

R2. eval(responseToOffer(Response, Specs,
    Client), VendorCert) :-
    issuer(rca_CA, 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 rca_CA. Moreover, if
    the PO is accepted a copy is sent to the auditor.

R3. eval(M, Cert) :- do(reject).

    All other requests are rejected.
```

Figure 2. A certified-contract example.

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 certified-contracts without any modification.

This is not true, however, for servers which need to be modified in order to deal with certified-contracts. (Details regarding server implementation are presented in Section 6.) Such servers may be maintained by organizations trusted by all parties involved in a contract, like large, trustworthy, financial institutions (for e.g., Visa and MasterCard), service providers (like Ebay, and AOL), or software providers (such as Sun, or Oracle, say). 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 its reputation too much to break contract terms purposely, by installing rogue servers. We mention that, since certified-contracts are signed, repositories may be maintained by any party involved or by a third party without creating security breaches.

4. Contract Annulment

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* (CRLs) maintained by trusted revocation servers. To revoke a certificate, the owner of the certificate, 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 disseminates the information [16].

Depending on the method used for disseminating revocation information one distinguishes between pull and push-based systems. In pull based systems, applications, which 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. We suggested earlier that contract annulment can be conveniently modeled by certificate revocation. Like for traditional certificates, both push or pull based mechanisms may be used to disseminate revocation notices.

However, unlike traditional certificates, CCs don't have an owner, and consequently revocation notices have to be issued by some designated authority. For now, we consider that the issuer of a CC is also authorized to send revocation notices. Since we have assumed that the issuer is a principal trusted by all parties involved in the contract, this ensures that a contract cannot be annulled arbitrarily, without the knowledge or consent of all parties.

5. 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 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, `rca_CA'` 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 06/15/2002 until 07/01/2002. 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 than \$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 `rca_CA` or `rca_CA'`.

Outdated Requests We are considering now 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 is validated provided it meets 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.

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

```

Preamble: type(contract).
         name(ford-rca)).
         version(2).
         validity([15,june,2002],[1,july,2002]).
         repository(http://trust.intertrust.com/).
         revocationServer(http://trust.intertrust.com/).

R1. eval(purchaseOffer( Specs , Amount ,
                      PaymentInfo , Vendor ) , ClientCert ) :-
    issuer( ford_CA , ClientCert ) ,
    role( manager , ClientCert ) ,
    do( accept ) .

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

R2. eval(purchaseOffer( Specs , Amount ,
                      PaymentInfo , Vendor ) , ClientCert ) :-
    issuer( ford_CA , 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 ford_CA
and (2) the Amount the agent is willing to pay for the mer-
chandise is less then $1000.

R3. eval( responseToOffer( Response , Specs ,
                          Client ) , VendorCert ) :-
    ( issuer( rca_CA , VendorCert ) or
      issuer( rca_CA' , 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
rca_CA or rca_CA' .

R4. eval( M , Cert ) :- do( reject ) .

All other requests are rejected.

```

Figure 3. Revised version of the contract-example.

amount offered does not exceed \$1,000. The question is: how should a purchase order arriving while the first version was valid, but processed after it was revoked, be treated? Here we chose to resolve the rule conflict by providing for a grandfather clause (see Rule $\mathcal{R}5$ in Figure 4).

```

R5. arbiter( version( 1 ) ,
            purchaseOffer( Specs , Amount ,
                          PaymentInfo , Vendor ) , ClientCert ) :-
    issuer( ford_CA , 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.)

6. Implementation

The system architecture, illustrated in Figure 5, relies on the existence of three trusted entities: revocation servers, contract repositories and generic policy-engines, called observers. Observers are trusted to verify certificates, interpret and carry out contract terms, and maintain the list of trusted contract issuers, TI. Under this scheme, an application server (e.g. Web, database, or e-mail server) has (at least) one 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 certified contract CC , and by a list (possibly empty) of subject certificates. Then the following steps are 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 CC has been signed by a principal belonging to TI , the list of trusted issuers.
- O checks if request R has arrived during the validity period inscribed in CC .

- 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 contract rules, the request, and the contract-version under which it was received.
- 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 the contract. Assuming that the repository maintains a newer version of the contract CC' , then O takes the following steps: First it checks whether, CC' itself has not been revoked. Second, it records the contract rules, and the contract-version under which the request was received.
- In case a valid version of the contract exists, then, for each subject certificate SC , accompanying the request, O checks that SC is valid by verifying that: (1) the signature is correct, (2) 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 contract version number under which the request was received.) If there is a rule sanctioning the request, then the observer carries 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 [6], 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 [7, 14, 1, 12]. However, to the best of our

knowledge, none of the frameworks proposed so far, embed contracts in certificates, nor do they deal with contract annulment or revision. We will briefly review here some of the general access control frameworks designed to support independently stated, evolving policies.

Blaze, Feigenbaum and Lacy [4] built a toolkit, called PolicyMaker, which can interpret arbitrary security policies. An agent receiving a request gives it for evaluation to PolicyMaker together with its specific policy, and the requester's credentials. Thus, this framework supports any number of policies, that can be updated/revoked at will by the agent in charge of a given PolicyMaker engine. However, the framework does not consider the case where the same policy is enforced by several, disparate policy engines, and thus does not provide means for assuring that all servers would update or revoke policies simultaneously. Another related attempt is Tivoli Policy Director [11], an impressive mechanism for controlling access to resources over geographically dispersed intranets and extranets. This framework, like ours, can support large sets of autonomous, dynamic policies. However, Tivoli is specifically designed for resource control, and it is not clear whether it can regulate applications, like e-commerce, which require control of inter-agent communication.

There are few researchers which, like us, propose to embed expressing various types of control policies in certificates. Theimer et al. [15] and Aura [2] proposed that delegation policies should be distributed as certificates; and Ioannidis et al. [10] 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. Thus, in these approaches, 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 certified-contracts 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.

8. Conclusion

We have argued that existent access control mechanisms cannot support adequately large sets of disparate, evolving

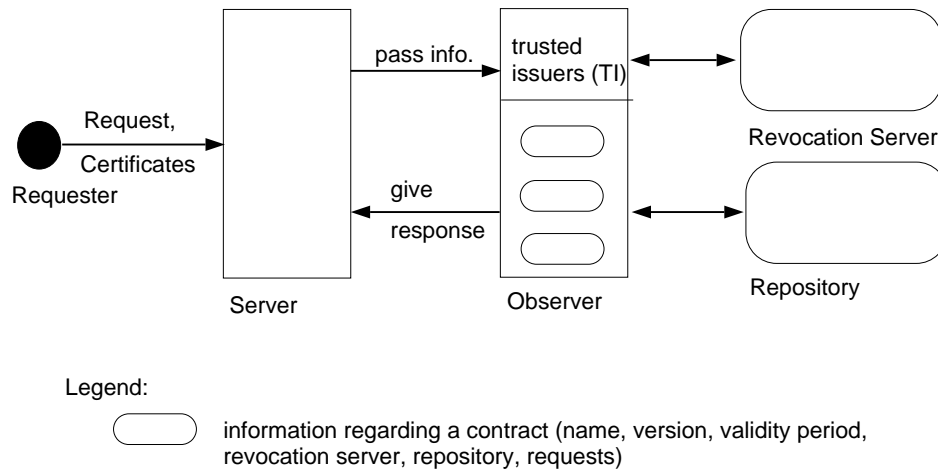


Figure 5. System architecture

contracts. To deal with this problem we proposed in this paper to certify contract terms, and to use the certificate infrastructure for contract management. This approach has several important benefits in B2B e-commerce context. First, one does not have to maintain a dedicated server for each contract (or set of contracts). 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.

But our presentation of the certified-contract concept is still wanting in some respects. In particular, we have assumed that contract revocation and update can be carried out only by the agent which issued the CC in question. This solution relies on the assumption that the trusted principal maintains its status and is available for the life time of the contract. Since this assumption might not always hold, it would be desirable to annul (revise) a contract without resorting to a single, hardwired trusted party. To deal with this issue we plan to explicitly embed into a CC information regarding additional agents which are allowed to call for an update or the annulment of the CC, and to investigate how the mechanism should be extended to support this type of extension. This, and other issues, will be discussed in a subsequent paper.

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] T. Aura. Distributed access-rights management with delegations certificates. In *Secure Internet Programming*, pages 211–235, 1999.
- [3] 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, Sept. 1998.
- [4] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.
- [5] P. Bonatti, S. D. C. 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. W. Consortium. Jigsaw - the W3C's web server. website: <http://www.w3.org/Jigsaw/>.
- [7] A. Dan, D. Dias, R. Kearny, T. Lau, T. N. Nguyen, F. N. Parr, M. W. Sachs, and H. H. Shaikh. Business-to-business integration with tpaML asnd a business-to-business protocol framework. *IBM Systems Journal*, 40(1):68–90, 2001.
- [8] Economist. E-commerce (a survey). pages 6–54. (The February 26th 2000 issue).
- [9] Economist. Riding the storm. pages 63–64. (November 6th 1999 issue).
- [10] 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.
- [11] G. Karjoth. The authorization service of Tivoli policy director. In *Proc. of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*, December 2001.
- [12] N. 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. 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 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.

- [15] M. Theimer, D. Nichols, and D. Terry. Delegation through access control programs. In *Proceedings of Distributed Computing System*, pages 529–536, 1992.
- [16] 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.