Certificate Management as Transaction Processing

Abstract

In the future it is expected that some form of public key infrastructure will be used to support critical applications such as financial transactions which require a high level of reliability, timeliness, and robustness. Current certificate management protocols are implemented on an ad hoc basis with no real guarantee as to their scalability or capability of meeting the required levels of performance. This paper examines the issue of certificate management from a transaction processing perspective, starting with the principles of transaction processing and applying them to fundamentals such as issuing certificates and CRLs, and then moving on to more advanced transaction-based operations such as the management of certificate status information across distributed and often unreliable systems. By building on the extensive body of knowledge which has been established through operating other transaction processing systems such as those used in the financial and airline industries, it is possible to create certificate management infrastructures which can provide certain guarantees which aren't present in current ad hoc approaches.

1. Certificate Management as Transaction Processing

It is envisaged that at some point in the future some form of certificate-based infrastructure will be used to provide authorisation and authentication facilities for high-value or critical transactions which are currently being carried out without the use of a PKI. These transactions are typically built on top of standard transaction processing (TP) principles which provide certain guarantees of reliability, timeliness, atomicity of transactions, and so on. Unfortunately the PKI mechanisms on which they will be relying are largely built on ad hoc principles, which reduce the reliability of the overall system to the reliability of the lowest common denominator as shown in the left half of Figure 1 in which a TP-based financial system relies on a non TP-based PKI. In order to provide the same performance guarantees as the transactions they are tied to, it's necessary for the certificate management process to also employ TP principles in order to avoid having it become the weak link in the process. Another way of putting this is that the certificate management mechanisms must be no less reliable than the TP mechanisms which rely on them. This alternative is shown in the right half of Figure 1.

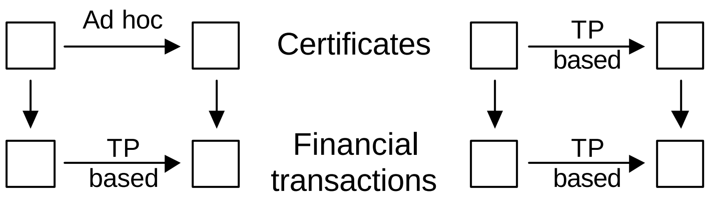

Figure 1: Backing financial transactions with ad hoc (left) and TP-based (right) certificate management

The transaction processing aspects of certificate management can be subdivided into two distinct function groups, the ones we know how to perform (things like issuing a certificate) and the ones we don't know how to perform (things like how to validate a signed data item, taking into account a certificate's suitability for its intended use, validation issues arising from revocation checking, and so on). The first part of this work covers the implementation details of the fundamental certificate management transactions, and the second part examines some of the further transactions which may be necessary to usefully employ certificates.

This paper is not intended to solve all certificate management problems (nothing can do that, since no-one is quite sure what those problems are). It isn't even meant to solve some certificate management problems, since the approaches to these are so many and varied that concentrating on a single solution invariably means that the dozen closely-related problems aren't quite addressed. Instead, this paper presents a series of fundamental building blocks on top of which it's possible to build arbitrary certificate management systems to which certain guarantees of performance can be attached.

A companion paper, "A Reliable, Scalable General-purpose Certificate Store" [1], examined the fundamentals of building a reliable, high-performance secure certificate store using COTS relational database (RDBMS) components. This paper extends the basic certificate store to provide a full-featured certificate management system using features which are either integrated into the RDBMS' or which can be easily added using standard commercial software components.

2. Introduction to Transaction Processing

Transaction processing represents a way of structuring the interactions between autonomous agents in a distributed system, and provides the equivalent of contract law for distributed systems (it has also been compared to a Christian wedding ceremony where the minister asks both sides if they agree to marry. If both sides agree the minister declares the transaction committed, otherwise it is aborted). Each transaction is a state transformation which obeys the so-called ACID properties (occasionally referred to as to passing the ACID test), where ACID stands for atomicity (the transaction is executed in an all-or-nothing manner), consistency (the transaction preserves the consistency of the database), isolation (each transaction is executed as if no other transactions were executing at the same time), and most importantly of all durability (the effect of the transaction, once completed, cannot be lost) [2][3]. ACID corresponds directly to the Clark-Wilson concept of a well-formed transaction, one which is constrained in such a way that it preserves the integrity of the data [4].

From a users point of view therefore, a TP system is expected to be completely reliable, execute the whole transaction (not just one part of it), and save the result of the transaction for later use. All this must occur in the presence of hardware and software errors, system and network failures (sometimes in the middle of processing the transaction), limited processor and storage resources, and a number of other pitfalls.

The first widely-used TP system was SABRE, developed in the early 1960's as a joint venture between IBM and American Airlines. At the time it was developed it was one of the biggest computer system efforts ever undertaken, and in the 40 years since the first system was installed has grown to connect more than 300,000 devices worldwide to handle 4,200 transactions per second.

Early TP systems typically involved thousands of terminals connected to large mainframes processing hundreds of thousands of transactions per day, but over the years as TP applications become more pervasive they migrated to smaller-scale systems and are now so commonly used not only in traditional applications such as banking, insurance, and stock trading but also for inventory control, retail and manufacturing, and electronic commerce that most users aren't even aware of their presence.

TP systems must meet a number of requirements. In order to meet scalability requirements, multiple transactions must be allowed to execute concurrently. However, since each transaction can involve multiple steps, one transaction can't interfere with the other. In addition, if one of the steps fail, any steps which have already been completed must be undone (rolled back) in order to make the whole transaction atomic. The TP system must also be able to produce authoritative records of completed transactions, and must often function in an environment where component systems are widely distributed.

In summary, a TP system must avoid producing partial results, handle concurrent operations, be scalable to handle high transaction volumes, never lose results, and avoid any downtime. This represents a considerable challenge which, if it can be met, makes a TP system ideally suited for certificate management.

2.1 Principles of Transaction Processing

It is important that a certificate management transaction be atomic (all-or-nothing) in that it executes completely or not at all. Consider the two situations shown in Figure 2, in which a CA issues a certificate to an end entity. In the first instance the CA issues a certificate and records the fact that it was issued, but experiences a server problem as it's being transferred to the certificate store. When the system recovers, the certificate is recorded as being issued by the CA but isn't actually issued, leaving the end entity unhappy. In the second instance, using a slightly different protocol in which the CA waits for an acknowledgement from the certificate store, the certificate is issued and entered into the certificate store, but is never recorded as being issued by the CA, leaving the CA both unhappy and in dire straights if a relying party suddenly discovers the supposedly non-existent certificate in active use. There are a large number of other variations on this theme, resulting in various types of data loss and/or inconsistency across the different parties, and the presence of more participants (for example end entity + RA + CA) only serves to complicate things further.

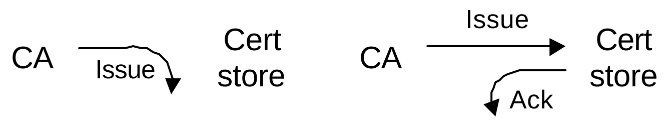

Figure 2: Certificate issuing problems, recorded but not issued (left), issued but not recorded (right)

There are many other types of failures which can occur, including lost updates (one update overwrites another while the first is completing), dirty reads (a read fetches data which hasn't been committed yet so that it may read a non-existent version of the data), phantoms (one or more data items are read based on a search condition after which an update occurs which changes the underlying data, resulting in different results if the same search is retried), and assorted others [5][6]. All of these problems need to be taken into account when managing certificates.

There have been examples in the past where implementers have tried to build fault-tolerant systems on ad hoc principles rather than using TP concepts. During this process they found that at a certain level they had hit a complexity barrier beyond which they couldn't push the ad hoc system any more. Re-implementing it using TP principles not only solved the problems, but lead to an overall gain in performance because the large number of ad hoc messages and disk I/O's needed to keep things running properly in the presence of failures were transformed into a much smaller number of TP-optimised messages. In terms of certificate management, the PKIX Certificate Management Protocol (CMP) [7] has already run into some of these sorts of problems [8] which are being fixed in an ad hoc manner as implementers gain experience with it [9]. CMP has to date only seen experimental use, so it's not known at which point the complexity barrier will occur.

Ensuring atomicity of transactions in distributed systems is a challenging problem since one or both systems can fail and recover while the transaction is taking place. The standard solution to this problem is the two-phase commit (2PC) protocol, in which each system affected by a transaction durably stores the transaction details before the transaction commits on any of the systems. This means that even if a system fails and recovers, it can commit the saved transaction (if it doesn't recover, the transaction is aborted). In either case at the end of the process all systems involved in the transaction have a consistent view of the transaction results. A outline of the 2PC process is shown in Figure 3 and analysed in great detail in any standard reference on databases or transaction processing. The diagram indicates how a successful transaction is handled, the handling of aborts is somewhat more complicated and is also well-covered in the literature [10][11][12].

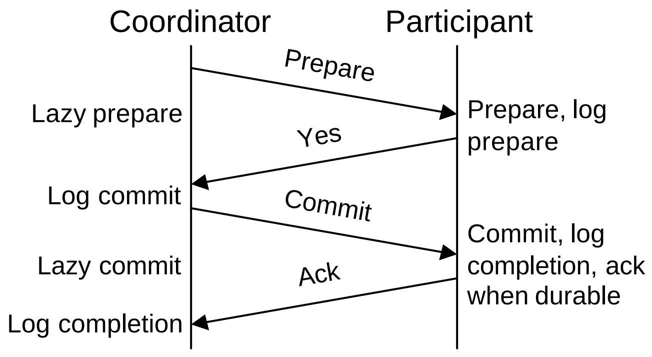

Figure 3: Two-phase commit

A helpful side effect of the use of TP is that it increases scalability by allowing the use of more systems to handle transactions, and availability by increasing the chances that one system will remain functional even if others fail, an issue which will be examined in more detail in a later section.

2.2 Transaction Priority

For some types of certificate information it's absolutely critical that they be distributed as quickly as possible, while for others it doesn't really matter if there's some delay before the information becomes available. Examples of the former class is certificate information used in financial protocols for which it's critical that it be distributed in a timely manner, examples of the latter class are the persona certificates which are handed out to anyone who can forge an email address (for this reason they have also been referred to as clown suit certificates because of their identification value and eventual validity if used in court [13]). The former type of certificate can be distributed with a higher priority than the latter type, in addition there are several different urgency classes in between the two which can be assigned varying priority levels if required.

Transactions have traditionally been broken down into two main classes, on-line transactions and batch transactions. On-line transactions typically execute very quickly and affect a very small portion of the database (examples being financial transactions), while batch transactions can take some time to execute and access a larger portion of the database (examples being complex queries and report generation). The natural match for these two transaction classes is to use online transactions to move an individual high-priority certificate and bulk transactions to move batches of low-priority certificates.

We can use a more fine-grained level of transaction priority than the simple distinction between online and batch by assigning different classes to different priority levels, where higher priority levels can bump lower priority ones. In theory it should be possible to apply ideas from the field of real-time databases to this problem, however these are concerned principally with the concept of executing a transaction by some fixed time, after which it is of no utility and can be discarded (that is, the transaction aborts). Typical real-time databases can schedule transactions based on the release time (the earliest time a transaction can start), the deadline (the time it must complete), and the estimated execution time [14][15][16], however these aren't terribly useful for the transactions which are likely to be used in certificate management and can even cause problems in some cases when almost-complete low-priority transactions are aborted because a high-priority transaction with a hard deadline has arrived (if this rather drastic step isn't taken, a priority inversion problem could occur in which a high-priority transaction is blocked while waiting on a lock held by a low-priority transaction). Although it may in some cases be useful to have this style of functionality, for example when certificate information must be distributed within a fixed, short deadline and an alarm raised if this fails, it's usually more useful to require that some certificate management transactions be carried out with all possible haste while others can be left to idle periods.

A better solution to the problem of certificate transaction priorities can be found in the field of multilevel secure (MLS) databases, ones which enforce the Bell-LaPadula security model for transactions. What this means in practice is that for reasons of non-interference, transactions at a lower security level must have a higher priority than those at a higher security level [17][18]. Although the issue of security levels is unimportant for certificates (which contain public keys and information), they provide a convenient prioritisation mechanism for transactions. A standard mechanism for providing prioritised transactions extends the typical transaction locking scheme by adding a conflict resolution mechanism which ensures that high-priority transactions aren't delayed by low-priority ones [19], since then a large number of addition prioritisation schemes with various performance tradeoffs have been proposed. MLS features have found their way into a number of standard database protocols.

Prioritisation options have begun to appear in various commercial RDBMS' and are a fairly standard feature of transaction processing systems such as BEA's Tuxedo, Transarc's Encina, IBM's MQSeries, and Microsoft's MSMQ. If necessary they can even be emulated (in a rather clunky fashion) in some RDBMS' which don't support prioritisation by assigning different priorities to the threads or processes which will be handling high- and low-priority transactions, although this loses some of the guarantees of TP-based priority management and if the transactions are valuable enough it would be better to use a real TP system with real prioritisation.

A hybrid approach which uses soft rather than hard deadlines can take advantage of the best features of TP-based and thread/process-priority-based priority management using techniques such as Adaptive Earliest Deadline (AED) [20], which in its simplest form divides incoming transactions into two groups, a HIT group for urgent transactions which can be processed quickly and a MISS or overflow group for lower-priority transactions which are processed as system load permits. Priorities can be adjusted dynamically to take into account current conditions on the system so that even if the system is very heavily loaded the end result is that more transactions are moved into the MISS group rather than being aborted. This technique was mainly designed as a means of handling overload conditions where the rate of arrival of high-priority transactions is greater than the rate of processing them, however it and various related overload management techniques are ideally suited to certificate management transactions since all we're concerned about is that some transactions are handled before others. In practice this isn't an issue today because the number of certificates issued each day by the largest CAs can be cleared by a full-scale TP system in a matter of seconds, but eventually it may become an issue in which case it's useful to know that the mechanisms to handle this are already in place.

2.3 Secure Transactions

Standard transactions can be made arbitrarily reliable in the face of various standard failures, but will still be vulnerable to malicious attacks which are explicitly designed to work around the reliability mechanisms. For example the principles used to guarantee transaction atomicity in the face of hardware and software failures won't work in the face of adversaries who can cause parts of the system to exhibit unpredictable behaviour and cause arbitrary failures. The former types of failure are generally referred to as benign failures (although they may not necessarily seem that way to affected users), while the latter types of failure are termed hostile or Byzantine failures [21]. Although standard TP principles will protect against benign failures, they need an extra level of armouring to secure them from hostile failures.

In many circumstances the certificates transactions will never be exposed to the possibility of hostile failures. For example if the systems participating in the transactions are on the same LAN or communicating over a private network then they will only have benign failures to deal with (any attacker who can get to the network will presumably be targeting the financial or other critical information directly rather than bothering with messing up the certificate management). On the other hand there will be times when certificate management transactions will need to travel over public networks such as the Internet. The type of security which is required in order to survive the hostile failures which can occur in these cases is common in many electronic commerce and digital cash protocols, although the details are usually present in a highly protocol-specific form which might guarantee such things as nonrepudiation (inasmuch as this is possible) and anonymity while only working for individual short messages. More general-purpose approaches towards securing generic transactions exist [22][23][24], however even these often provide unnecessary functionality and share with the more specialised protocols a relatively high overhead, both cryptographically and in the number of messages exchanged, for each transaction.

This type of per-message security represents an unnecessary feature for standard TP protocols, since they already handle benign failures and only need to be protected against hostile ones. In order to do this, we need to provide a layer of integrity protection for the protocol which prevents an attacker from tampering with communications in a manner designed to cause failures. If this type of extra protection is required, it can be readily added using standard integrity-protection-only SSL or IPsec tunnelling, features which are already present in many RDBMS or TP systems and some operating systems. It may in addition be desirable to add confidentiality protection to prevent trivial traffic analysis in case an attacker can cause problems by interrupting the message flow at inopportune moments, for example in order to force excessive numbers of transaction aborts and rollbacks resulting in a relatively subtle denial-of-service attack.

3. Basic Certificate Management Transactions

Now that we have a general model for handling certificate management transactions, we need to look at the individual transactions which make up the certificate management process. In terms of bulk certificate transactions (that is, heavyweight operations such as certificate issuing which will be performed repeatedly, as opposed to one-off certificate retrievals and validity checks), the flow of operations is shown in Figure 4. The certificate management process begins when an end entity (EE) submits a request for a certificate, either to a CA or to a registration authority (RA) which has a trusted relationship with the CA and acts as a proxy for some CA operations, performing actions such as user identity verification on its behalf (a typical RA would be the HR division of a corporation, which is far more capable of checking user details than a CA located on the other side of the planet).

If an RA is being used, the RA could forward the EE's certificate request in the form of a short-term RA certificate to the CA, otherwise the CA processes the EE's certificate request directly. Eventually the CA issues a certificate and either passes it back to the EE or makes it available via some other, usually implementation- and situation-specific means. The exact mechanisms involved in operating the various certificate data stores are covered in more detail in the companion paper mentioned earlier.

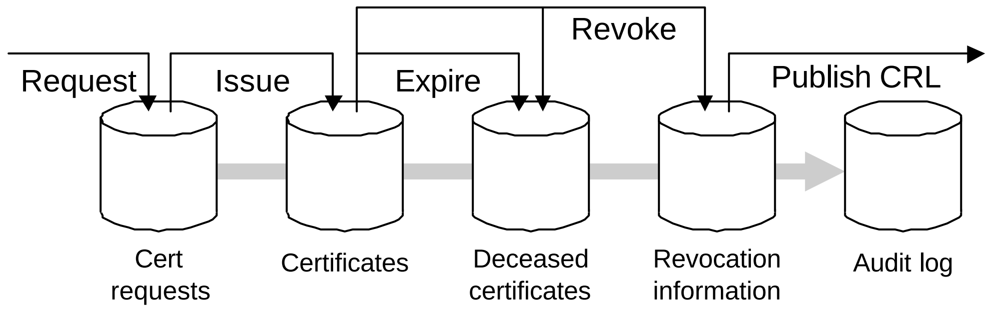

Figure 4: Typical certificate management transaction flow

In addition to issuing certificates, the CA is also charged with issuing revocations (CRLs) for certificates on behalf of the owning EE. It should be mentioned here that CRLs don't really work (the reasons for this are many and varied and are examined in more detail in a later section), but given that they appear to be an expected fixture in a PKI we have to generate them, at least for now. TP-based alternatives to CRLs will be discussed later on.

```
read certificate request from store
create certificate from request

start
   delete certificate request from store
   record certificate as issued
   write certificate to store
commit
```

Figure 5: Certificate issue transaction

The first transaction to consider is that of issuing a certificate, in the simplest case directly to the end entity. The basic certificate issue transaction is shown in Figure 5. Because of the ACID properties of the TP system, we know that once the transaction has completed either the certificate request will have been converted into a certificate and the auditing records updated, or it'll remain in its original form in the certificate request store. Note that the ephemeral parts of the operation (turning the certificate request into a certificate) have been left outside the actual transaction, since they don't result in any permanent changes being made. If there's some form of auditing performed on CA private key operations so that a signature operation must result in either the appearance of a certificate or a record of an aborted transaction then it may be necessary to move at least the certificate signing operation into the transaction as shown in Figure 6.

```
read certificate request from store
build certificate data from certificate request

start
   delete certificate request from store
   sign certificate
   record certificate as issued
   write certificate to store
commit
```

Figure 6: Modified certificate issue transaction

Issuing a certificate via an RA is a slight variation on this theme in which the RA first executes the transaction shown in Figure 5 or Figure 6 to convert the end entity's certificate request into a short-duration RA certificate or some equivalent certificate object, and the CA executes a transaction to convert the short-duration RA certificate into a standard certificate. This process is shown in Figure 7.

```
read short-term RA certificate from store
create certificate from short-term RA certificate

start
   delete short-term RA certificate from store
   record certificate as issued
   write certificate to store
commit
```

Figure 7: Certificate issue transaction involving an RA

The only remaining operation is certificate revocation, which is shown in Figure 8. This operation is broken down into two discrete steps, revoking an individual certificate, and at some later point issuing a CRL containing zero or more revocations. Revoking a certificate involves moving the certificate from the certificate store to the deceased certificates store and updating the CRL store to reflect the fact that the revocation has taken place. Unlike the previous transactions, this one affects three separate tables. A slight variation of this is the certificate expiry transaction, which simply moves a certificate from the active to the deceased certificate store.

```
start
   delete certificate from store
   add certificate to deceased certificates store
   add revocation entry to CRL store
commit
```

Figure 8: Certificate revocation transaction

Issuing a CRL, which consists of a simple linear table scan to read revocation information, isn't in itself a TP-type operation, but it does provide its own problems due to the way it can interact with other TP operations. Although the CRL store has been structured to make the scan operation as efficient as possible, there's still the problem that either the entire table must be locked while the scan takes place (blocking all revocation transactions until it completes), or that the issued CRL doesn't accurately reflect the state of the store due to new revocations being processed as the scan is taking place. In more general terms, we need to be able to act on data as it exists at time *t* even though the operation spans from *t* to *t* + ?, during which time other transactions are simultaneously updating the data (in the CRL case we need to know which revocations have been issued at time *t*).

The existence of an operation which can lock an entire table is particularly troublesome in the case of the certificate revocation transaction, since it updates all three of the main tables which will be used for CA operations. Without careful attention to locking details, a locked CRL table will also back up transactions on the other tables, causing either a general loss of performance or a halt to all transactions until the CRL table lock is released and the backlog clears.

This issue is a standard TP problem, and there are a number of standard solutions to it. The simplest one (which, however, only applies to the CRL table because of the way it is structured) takes advantage of the fact that locking a node of a B-tree implicitly locks all the nodes below it because all search operations below that node need to pass through the locked node [25], so that (due to the fact that all entries are clustered by the issuing CA name) simply performing the scan results in consistent data being returned. A more rigorous variant of this, which is typically performed automatically by the database, is key-range locking, in which the rows covered by a key value (helpfully provided in the SQL query by the `WHERE` clause) are locked but others are left open.

Unfortunately this is a somewhat crude mechanism which still blocks access to all revocations for a given CA while the scan takes place, it requires careful design of the underlying database table to work, and it doesn't work at all for some RDBMS' (for example MS SQL Server) which implement clustered indices in peculiar ways.

A much more rigorous approach is to make use of a TP system to provide multi-version data, in which updates don't overwrite any existing data but instead add a new version of the data alongside the existing version, so that each data item consists of a sequence of entries, one for each update which was applied. This means that it's possible to both scan a table and perform updates on it while still returning consistent results for the scan. More generally, it's possible to obtain data as it was at time *t* even if the scan is taking place long after *t* has passed.

A slight variation on this, which doesn't use any locking at all, is to run the scan in snapshot mode, in which the query manager returns the last version of the data which was committed at the time the query was executed. Again, this allows updates even as the query is executing, with no contention at all due to locking.

A final alternative, which also solves a number of other problems, is to never issue any CRLs at all. This is discussed in more detail later on.

3.1 Extended Certificate Management Transactions

Now that we've examined the basic certificate management transactions we need to look at some of the more advanced types of transactions which are needed to maintain the state of the certificate store. One of the simplest ones is the general housekeeping process which moves expired certificates from the currently active store to the deceased certificate store. This is a relatively straightforward operation which consists of a variation of the certificate revocation transaction which performs a `SELECT` on the certificate expiry date and relocates all certificates in the result set rather than explicitly revoking a certificate.

A related transaction is used to revoke a group of certificates when a CA key is revoked. This is an operation which has the potential to require quite a number of transactions if the CA has issued a large number of certificates, but fortunately a CA key revocation is such a rare occurrence that in practice it's unlikely to have much of an effect. As with the certificate expiry operation, the process involves selecting all certificates issued by the CA certificate which is being revoked and moving them to the expired certificate store.

A number of additional certificate transactions may also be required for the day-to-day running of the system. Since the underlying certificate store is an RDBMS, these are fairly easily implemented in a few lines of standard SQL.

4. Distributed Transactions

The previous section covered the types of transactions involved in localised certificate processing such as issuing a certificate to a user. Complementing these types of transactions are ones which occur on a larger scale and are used to manage certificate information across distributed systems. The former class of transactions are known as local transactions, the latter are global transactions. In general we can extend the local transaction scheme to a global one by employing a distributed two-phase commit of the type show in Figure 9. The protocol begins by having a central coordinator send a prepare message to each participant. The participants then reply indicating whether they are prepared to commit the transaction, if all participants vote to commit then the coordinator sends the commit (or abort) message and the transaction is committed or aborted as required. There are many variants of the distributed 2PC which include a variety of optimisations such as ones which assume that transactions will usually commit (presumed-commit) or abort (presumed-abort), ones which allow direct communication between participants to eliminate the potential bottleneck of the single coordinator, replication schemes to handle failure of a coordinator, and various other optimisations and fault-tolerance measures [26][27][28].

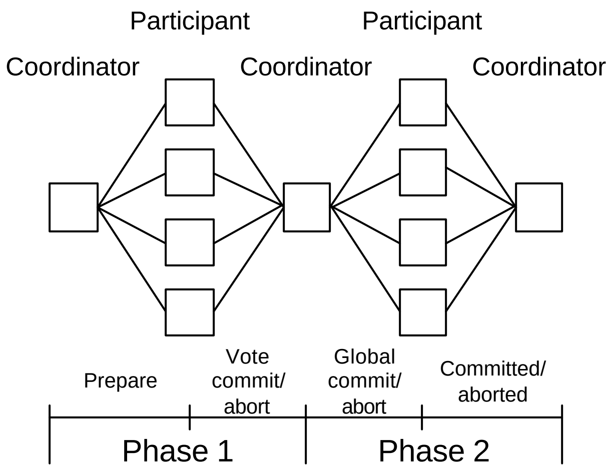

Figure 9: Distributed two-phase commit

The advantage of having multiple distributed copies of data is that it increases reliability because of the greater probability of at least one server being up, and decreases access delays because there's less contention for access to an individual server and because it's possible to fetch the data from a closer (in terms of network time) server [29][30].

Distributed databases fall into two main classes, multidatabases in which each local database contains a subset of the overall data, and true distributed databases in which each local database mirrors the state of every other local database. Multidatabases are used primarily where a collection of pre-existing autonomous databases needs to be linked without updating applications which assume a particular database schema and introduce all sorts of complications such as schema translation from what's really there to what's expected to be there and schema homogenisation which is an even more interesting problem. These won't be considered further here since it's unlikely that there's any large installed base of RDBMS-based certificate stores for which there's a requirement that they be connected in a multidatabase. This leaves true distributed databases, in which each local database represents a local image of the global database.

4.1 Distributed Databases

When updating a distributed database in the presence of unreliable hardware, software, and communications links, we need to decide what to do if a failure occurs. In the discussion of distributed 2PC above the transaction was presented as an all-or-nothing affair in which all participants ended up with a consistent view of the data (known as one-copy equivalence) at the expense of having all participants abort the transaction if a single one voted for it. In some cases it may be desirable to allow the transaction to be committed in the presence of certain types of failures provided that the failed system is brought back into the same state as the remaining participants before it resumes normal operation.

Handling this situation becomes a standard database replication problem which has been studied closely for many years and for which a number of solutions exist [31][32]. The one-copy equivalence criterion is usually enforced using a protocol called Read One Write All (ROWA) which converts a logical read to a read on any one of the replicas and a logical write to a write on all replicas. A far more relaxed criterion than straight ROWA is Read One Write All Available (ROWA-A) where writes are executed on all available copies and the ones who were unavailable catch up when normal operation is restored.

There are many variations of this type of protocol, of which the most general are voting schemes in which a write operation has to obtain a sufficient number of votes in order to commit. In the most extreme case this becomes ROWA, otherwise the required quorum value can be adjusted to match the particular requirements.

5. Distributed Certificate Management Transactions

Having covered the principles of distributed databases and the distributed transactions which are used to handle them, we can apply these principles to the process of certificate management. The basic entity involved in this process is a community, a collection of certificate-processing systems and relying parties tied together by TP-based certificate management. Communities are likely to be small (relative to the size of the entire Internet) and tied together by a common interest or policy requirements, for example the automated clearing house (ACH) network is an example of such a community which is tied together by both a very stringent set of operating requirements and the operating rules of the ACH system. Another example is Identrus, which is aimed at providing real-time certificate status information (along with many other things) to members of its community.

For communities with a larger scope such as the entire Internet, the only certificates which are likely to be used at this level are clown-suit ones which no-one really cares about, in the sense that there's no value attached to them so there's nothing to go to court over if there's a problem. A TP-based community probably won't scale to this level, but then again it's unlikely that there'll be much call for anything at this scale (even the most grandiose non-clown-suit PKI schemes, woven together with a network of cross-certification and bridge CAs, probably wouldn't aspire to this level of universal coverage).

The advantage which a TP-based community has is that, thanks to the properties of the TP systems described earlier, it guarantees that each member of the community has a completely consistent view of a certificate. If one member of the community determines that a certificate is currently valid, they know that every other member of the community will share that view for that given point in time. Furthermore, if the owner of a certificate needs to know whether others in the community currently regard their certificate as valid they can perform a simple inquiry which unambiguously indicates how every other member in the community sees the certificate at that time (historical queries, which indicate how the certificate was perceived by the community five minutes ago, are covered in a later section).

A similar concept has been hypothesised in the form of a trusted directory which stores known-good certificates for reference by relying parties (assuming they can figure out whether they're supposed to be looking for a certificate or userCertificate or cACertificate or crossCertificatePair or whatever other tag the directory might store certificates under). This approach, which relies on the existence of a reliable, scalable distributed X.500-style directory system appears to exist only in theory, and without major restructuring of the underlying mechanisms probably isn't capable of supplying the guarantees of the TP-based system presented here.

5.1 Problems with Certificate Revocation

As has already been mentioned earlier, certificate revocation is something which doesn't really work, particularly when attempts are made to implement it in the manner envisaged in X.509. The main reason for this is that the use of CRLs violates the cardinal rule of data-driven programming which is that once you have emitted a datum, you can't take it back. Viewing the certificate issue/revocation cycle as a proper TP-style transaction with ACID properties, the certificate issue becomes a `PREPARE` and the revocation a `COMMIT`, however this means that nothing can be done with the certificate in between the two because this would destroy the ACID properties of the transaction. Allowing for other operations with the certificate before the transaction has been committed results in non-deterministic behaviour, with the semantics of the certificate being reduced to a situation described as "This certificate is good until the expiration date, unless you hear otherwise" [33] which is of little value to relying parties.

CRLs have many other problems which make them difficult to work with and unreliable as a certificate status propagation mechanism. The general problem which needs to be solved, and the one which CRLs don't really solve, is that critical applications (which usually mean ones where a lot of money is involved, but can be extended arbitrarily to cover whatever the relying parties consider important enough) require prompt revocation from a CRL-centric world view, or require real-time certificate status information from a more general world view. Attempting to do this with CRLs presents a number of problems.

The general idea behind a CRL is based on the credit-card blacklists which were used in the 1970's in which credit card companies would periodically print booklets of cancelled card numbers and distribute them to merchants who were expected to check any cards they handled against the blacklist before accepting them. The same problems which affected credit-card blacklists then are affecting CRLs today: the blacklists aren't issued frequently enough to be effective against an attacker who has compromised a card or private key, they cost a lot to distribute, and they are easily rendered ineffective through a denial-of-service attack (to ensure that your card is never blocked through a blacklist, make sure the blacklist is never delivered).

When a CA issues a CRL, it bundles up a blacklist of revoked certificates along with an issue date and a second date indicating when the next blacklist will become available. A relying party which doesn't have a current CRL is expected to fetch the current one and use that to check the validity of the certificate. In practice this rarely occurs because users and/or applications don't know where to go for a CRL, or it takes so long to fetch and check that users disable it (in one widely-used application enabling CRL checking resulted in every operation which used a certificate being stalled for a minute while the application groped around for a CRL, after which it timed out and processing continued as before), or they simply can't be bothered and put things off until they can't do anything any more (assuming they even know the significance of a revoked certificate and don't just interpret the failure to perform as an error in the application) [34].

Moving beyond the user interface problems (which will inevitably affect any certificate status mechanism, no matter how sophisticated), there are a number of practical issues with CRLs. In order to guarantee timely status updates, it's necessary to issues CRLs as frequently as possible, however the more often a CRL is issued the higher the load on the server which holds the CRL, on the network over which it is transmitted, and (to a lesser extent) on the client which fetches it. Issuing a CRL once a minute provides moderately timely revocation (although still not timely enough for cases such as Federal Reserve loans where interest is calculated by the minute) at the expense of a massive amount of overhead as each relying party downloads a new CRL once a minute. This also provides a wonderful opportunity for a denial of service attack on the CA in which an attacker can prevent the CRL from being delivered to others by repeatedly asking for it themselves. On the other hand delaying the issuing to once an hour or once a day doesn't provide for any kind of timely revocation.

A second problem with CRLs with built-in lifetimes is that they all expire at the same time, so that every relying party will fetch a new CRL at the same time, leading to huge peak loads whenever the current CRL expires. One ad hoc solution to this problem is to stagger CRL expiry times for different classes of certificates so that they don't all expire at the same time, although scheduling the CRL times while still providing some form of timeliness guarantee is sure to prove a tricky exercise. Another approach is to over-issue CRLs so that multiple overlapping CRLs exist at one time, with a relying party who fetches a CRL at time n being fed a CRL which expires at time $n + 5$ and a relying party who fetches it at time $n + 2$ being fed one which expires at time $n + 7$. Assuming that CRL fetching patterns follow a certain distribution, this has the potential to lighten the peak loads on the server somewhat [35] at the expense of playing havoc with CRL semantics.

Yet another approach is to segment CRLs based on the perceived urgency of the revocation information, so that a CRL with a reason code of "key compromise" would be issued more frequently than one with a reason code of "affiliation changed". This segmenting doesn't reduce the request rate but can reduce the amount of data transferred so that it takes less time to service an individual request. Segmenting CRLs has the side-effect that it introduces security problems since an attacker who has performed a key compromise can use the compromised key to request that a revocation for it be placed in a low-priority CRL, ensuring that the key is both regarded as safely revoked by the CA but at the same time will still be valid until the next low-priority CRL is issued at the end of the day or week [36]. Solving this problem requires very careful protocol design and extensive amounts of checking and safeguards at every step of the revocation process.

Yet another approach involves delta CRLs, which have one large long-term CRL augmented by a number of smaller, short-term CRLs which modify the main CRL. There appears to be little real-world experience in the use of delta CRLs, although discussion on PKI mailing lists indicates that attempts to implement them will prove to be an interesting experience. Beyond this there exist even more approaches to the problem, tinkering with revocation mechanisms is a popular pastime among PKI researchers.

One possible solution to this problem is used in SET, which takes advantage of the fact that certificates are tied to credit cards to avoid the use of CRLs altogether. SET cardholder certificates (which are expected to be invalidated relatively frequently) are revoked by revoking the card which they are tied to, merchant certificates (which would be invalidated far less frequently) are revoked by removing them from the acquiring bank's database, and acquirer payment gateway certificates (which would almost never be invalidated) are short-term certificates which can be quickly replaced. This process takes advantages of existing mechanisms for invalidating certificates, or designs around the problem so that revocation in the manner expected of other PKIs is never needed. A similar type of scheme which designs around the problem is used in Account Authority Digital Signatures (AADS), a simple extension to existing account-based business infrastructures in which public keys are stored on a server which handles revocation by removing the key. To determine whether a public key is valid, the relying party fetches it from the server [37][38].

5.2 Certificate Status Checking

In the presence of a distributed certificate store which contains known-good certificates, the problem of certificate status checking is greatly simplified. Since one of the properties of the distributed store is that it provides a consistent view of the status of a certificate, anyone in the community can obtain a definitive response to a query about a particular certificate with the guarantee that everyone else in the community will agree with the assessment. A response to a query about a certificate is a simple boolean value, "The certificate is valid right now" or "The certificate is not valid right now".

In comparison to this simple yes/no response, methods such as the Online Certificate Status Protocol (OCSP) [39] (which relies on CRLs and ad hoc certificate management mechanisms) must resort to the use of multiple non-orthogonal certificate status values because they can't provide a definitive answer to a query about a certificate. In the case of OCSP the possible responses to a query are "good", "revoked", and "unknown", where "good" doesn't necessarily mean "not revoked" and "unknown" could mean anything from "this certificate was never issued" to "it was issued but I couldn't find a CRL for it" (this vagueness is at least partially the fault of a CRL-based certificate status mechanism, since a CRL can only provide a negative result so that the fact that a certificate is not present in a CRL doesn't mean that it was ever issued). For some OCSP implementations the "I couldn't find a CRL" response may be reported as "good", or will be interpreted as "good" by relying parties since it's not the same as "revoked" which is assumed to be "not good" (opinions on the exact semantics of the various responses vary somewhat among implementers).

A slightly different approach is taken in the Simple Certificate Validation Protocol (SCVP) [40] which either treats the server as a dumb repository or as a full certificate chain validation system in which the relying party submits a collection of certificates and optional ancillary information such as policy information, and the SCVP server indicates whether the chain can be verified up to a trusted root. This service seems to be aimed mostly at thin clients who can't perform any path validation themselves, although there is some debate among proponents as to its merits relative to approaches like OCSP. A more stealthy approach is taken in the Data Validation and Certification Server Protocols (DVCS) [41] which provides as part of the protocol a facility more or less identical to SCVP but which has escaped controversy by presenting itself as a third-party data validation mechanism rather than a certificate status protocol.

Alongside the warring OCSP and SCVP camps and DVCS stealth approach, a number of other certificate status protocols have appeared and disappeared over time, including the Integrated CA Services Protocol (ICAP) [42], the Real-Time Certificate Status Protocol (RCSP) [43], the Web-based Certificate Access Protocol (WebCAP) [44], the Open CRL Distribution Process (OpenCDP) [45], the Directory Supported Certificate Status Options (DCS) [46], and many, many others (the protocol debate has been likened to religious sects arguing over differences in dogma [47]). Since the author believes in freedom of religion, this paper doesn't contain any requirements for a particular protocol to access the certificate information, although some recommendations will be provided where appropriate.

It should be mentioned at this point that this section is not intended to present yet another certificate status protocol (there are already more than enough of those to go around). Instead, it describes an application of the TP-based certificate management system described earlier to the problem of determining certificate status information across a distributed environment and in the presence of arbitrary hardware and software failures. Figure 10 represents an example of this, illustrating a typical community during a certificate status update. Initially all hosts which maintain status information (except E, which is down at the moment), maintain a standard view of the validity of a certificate which is presented to relying parties (the thin black lines) as shown in the leftmost portion of the diagram.

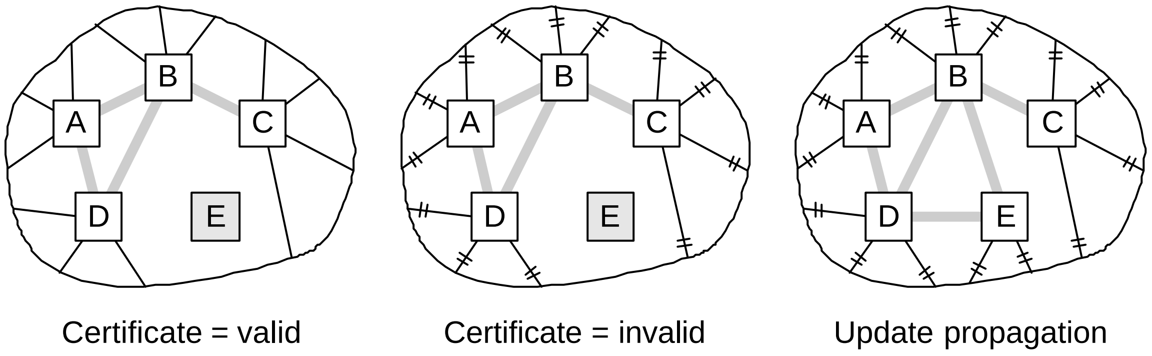

Figure 10: Community before update (left), after update (middle), update propagation (right)

The situation after a ROWA-A update is shown in the centre portion of the diagram, where all hosts now agree that the certificate is invalid and all relying parties are again presented with the unified view of the new status. Finally, the downed host rejoins the community and the situation shown on the right-hand portion of the diagram ensues, with all present hosts still presenting the unified view of the certificate status. The entire update process, which in X.509 terms is equivalent to propagating a CRL to all relying parties simultaneously, requires a single transaction across the hosts A-D and an update for host E when it comes back online.

The previous section discussed the problems involved in pull CRLs, where obtaining up-to-date certificate status information could entail down loading a CRL with thousands of entries every few minutes, delta CRLs and other kludges notwithstanding. This represents an extraordinarily inefficient means of disseminating revocation information, since with (say) a thousand-entry CRL it's necessary for a relying party to repeatedly fetch 999 irrelevant entries in order to obtain status information for the one certificate which they care about, at the same time placing a heavy load on the CA which sources the CRLs [48]. In contrast, a TP-based approach amortises the load placed on the system by only distributing the revocation information for the one certificate being revoked, and only distributing it once.

Since the community is based on the system described in the companion paper mentioned earlier, it's possible to go much further than this and also provide historical certificate status information. This type of information can't be provided by conventional means such as CRLs or OCSP since they can only convey information about when a certificate was revoked (that is, they can only report the status as "good" or "revoked at time *t*", which doesn't indicate whether the certificate was ever issued and in circulation). Because the certificate stores which underlie the community record precise timestamp information which, due to the nature of the TP mechanisms, is synchronised across the entire community, it's possible to respond to historical queries in the same manner as with standard queries, namely that the entire community agrees that at time *t* a given certificate was valid and active in the community, or that it was not valid and active.

The mechanisms which can be used to communicate this information to relying parties are many and varied. Obviously the standard protocols such as OCSP or SCVP can be pressed into use, although they are incapable of properly representing the full range of facilities provided by the TP approach for the reasons given above. Alternatively, more lightweight approaches such as ones designed for use in thin clients may be used. A simple and very generic approach for high-volume applications is to submit a hash of the certificate to be queried over a pre-established secure channel (for example IPsec or SSL or a direct LAN connection), receiving as reply the status of the certificate. This simple mechanism can be garnished to taste to handle historical queries and other bagatelles. Where queries are more infrequent, a per-query authentication mechanism such as a digital signature or MAC or similar mechanism can be used for query and response authentication [49].

The advantage of supplying a hash rather than a full certificate (apart from the obvious ones such as the bandwidth saved and the fact that a certificate hash provides a more unambiguous means of identifying a certificate than alternatives such as an issuer name or other not necessarily unique identifier), is that it doesn't leak potentially confidential information about certificate contents, a problem which in the X.500 world requires workarounds like border directories which contain sanitised versions of what's actually available. Since the only item of interest is "Is the certificate with this hash valid/still valid" (with a possible side order of "When did it become valid/invalid"), there's no real need to ship a full certificate. As the companion paper showed, it's possible to answer such queries extremely efficiently using an appropriately configured RDBMS.

6. Conclusion

This paper has presented a new approach of viewing certificate management operations in terms of traditional transaction processing mechanisms which can be used with certain guarantees of performance and resilience in the presence of failures. Applying TP methods allows the certificate management process to be rigorously designed to provide extensive data control and availability, high multi-user throughput, and predictable, fast response times using established practices and software. When used within a community of relying parties, the TP-based system provides a unified, consistent view of a certificate's status across the entire community, avoiding the drawbacks of existing approaches such as ones based on CRLs. Finally, since the certificate management system is based on proven, well-established TP technology, it can be implemented using standard components without requiring the development of complex custom approaches aimed at handling fault-tolerance, reliability, and scalability issues.

7. References

[1] "A Reliable, Scalable General-purpose Certificate Store", Peter Gutmann, *Proceedings of the 14th Annual Computer Security Applications Conference*, December 2000, to appear.

[2] "Transaction Processing: Concepts and Techniques" Jim Gray and Andreas Reuter, Morgan Kaufmann, 1993.

[3] "Atomic Transactions", Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete, Morgan Kaufmann, 1994.

[4] "A Comparison of Commercial and Military Computer Security Policies", David Clark and David Wilson, Proceedings of the 1987 IEEE Symposium on Security and Privacy, April 1987, p.184.

[5] "A Critique of ANSI SQL Isolation Levels", Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, May 1995, p.1.

[6] "Semantic ACID Properties in Multidatabases Using Remote Procedure Calls and Update Propagations", Lars Frank and Torben Zahle, *Software — Practice and Experience*, **Vol.28**, **No.1** (January 1998), p.77.

[7] "Internet X.509 Public Key Infrastructure: Certificate Management Protocols", RFC 2510, Carlisle Adams and Stephen Farrell, March 1999.

[8] "CMP Interoperability Testing: Results and Agreements", Robert Moskowitz, `draft-moskowitz-cmpinterop-00.txt`, June 1999.

[9] "Internet X.509 Public Key Infrastructure: Certificate Management Protocols", Carlisle Adams and Stephen Farrell, `draft-ietf-pkix-rfc2510bis-01.txt`, July 2000.

[10] "Transaction Processing: Concepts and Techniques" Jim Gray and Andreas Reuter, Morgan Kaufmann, 1993.

[11] "Atomic Transactions", Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete, Morgan Kaufmann, 1994.

[12] "Principles of Transaction Processing", Philip Bernstein and Eric Newcomer, Morgan Kaufman Series in Data Management Systems, January 1997.

[13] "Re: Purpose of PEM string", Doug Porter, posting to `pem-dev` mailing list, message-ID `93Aug16.003350pdt.13997-2@well.sf.ca.us`, 16 August 1993.

[14] "On Real-Time Databases: Concurrency Control and Scheduling", Philip Yu, Kun-Lung Wu, Kwei-Jay Lin and Sang Son, *Proceedings of the IEEE*, **Vol.82**, **No.1** (January 1994), p.140.

[15] "Real-time Databases: Issues and Applications", Bhaskar Purimetla, Rajendran Sivasankaran, Krithi Ramamritham, and John Stankovic, *Advances in Real-Time Systems*, Prentice-Hall, 1995, p.487.

[16] "Real-Time Database Systems: Issues and Applications", Azer Bestavros, Kwei-Jay Lin, and Sang Son (editors), Kluwer International Series in Engineering and Computer Science, Vol.396, Kluwer Academic Publishers, May 1997.

[17] "Secure Transaction Processing in Secure Real-Time Database Systems", Binto George and Jayant Haritsa, *Proceedings of the 1997 ACM SIGMOD International Conference on the Management of Data*, ACM, May 1997, p.462.

[18] "Advanced Transaction Processing in Multilevel Secure File Stores", Elisa Bertino, Sushil Jajodia, Luigi Mancini, and Indrajit Ray, *IEEE Transactions on Knowledge and Data Engineering,* **Vol.10**, **No.1** (January/February 1998), p.120.

[19] "Scheduling Real-time Transactions: A Performance Evaluation", Robert Abbott and Hector Garcia-Molina, *ACM Transactions on Database Systems*, **Vol.17**, **No.3** (September 1992), p.513.

[20] "Earliest Deadline Scheduling for Real-Time Database Systems", Jayant Haritsa, Miron Livny, and Michael Carey, *Proceedings of the 12th Real-Time Systems Symposium (RTSS'91)*, IEEE Computer Society Press, 1991, p.232.

[21] "Atomic Broadcast: From Simple Diffusion to Byzantine Agreement", Flaviu Cristian, Houtan Aghill, Ray Strong, and Danny Dolev, *Proceedings of the 15th Annual International symposium on Fault-tolerant Computing*, IEEE Computer Society Press, June 1985, p.200.

[22] "Verifiable Transaction Atomicity for Electronic Payment Protocols", Lei Tang, *Proceedings of the 16th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, 1996, p.261.

[23] "A Protocol for Secure Transactions", Douglas Steves, Chris Edmondson-Yurkanan, and Mohamed Gouda, *Proceedings of the 2nd Usenix Workshop on Electronic Commerce*, November 1996.

[24] "Building Blocks for Atomicity in Electronic Commerce", Jiawen Su and Doug Tygar, *Proceedings of the 6th Usenix Security Symposium*, July 1996, p.97.

[25] "Concurrency Control and Recovery in Database Systems", Philip Bernstein, Vassos Hadzilacos, and Nathan Goodman, Addison-Wesley, 1987.

[26] "Distributed Databases: Principles and Systems", Stefano Ceri and Giuseppe Pelagatti, McGraw-Hill, 1984.

[27] "Distributed and Multi-Database Systems", Angelo Bobak, Artech House, 1995.

[28] "Revisiting Commit Processing in Distributed Database Systems", Ramesh Gupta, Jayant Haritsa, and Krithi Ramamritham, *Proceedings of the 1997 ACM SIGMOD International Conference on the Management of Data*, May 1997, p.486.

[29] "Recovery Concepts for Data Sharing Systems", Erhard Rahm, *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, IEEE Computer Society Press, June 1991, p.368.

[30] "The Performance of Two-phase Commit Protocols in the Presence of Site Failures", M.Liu, D.Agrawal, and A.El Abbadi, *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, IEEE Computer Society Press, June 1994, p.234.

[31] "Replication Techniques in Distributed Systems", Abdelsalam Helal, Abdelsalam Heddaya, and Bharat Bhargava, Kluwer Academic Publishers, 1996.

[32] "Principles of Distributed Database Systems (2nd ed)", M. Tamer Ozsu and Patrick Valduriez, Prentice Hall, 1999.

[33] "Can We Eliminate Certificate Revocation Lists", Ronald Rivest, *Proceedings of the 2nd International Conference on Financial Cryptography (FC'98)*, Springer-Verlag Lecture Notes in Computer Science No.1465, February 1998, p.178.

[34] "Making Netscape Compatible with Fortezza— Lessons Learned", George Ryan, *Proceedings of the 22nd National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1999, CDROM distribution.

[35] "A Model of Certificate Revocation", David Cooper, *Proceedings of the 15th Annual Computer Security Applications Conference (COMPSAC'99)*, December 1999, p.256.

[36] "A Closer Look at Revocation and Key Compromise in Public Key Infrastructures", David Cooper, *Proceedings of the 22nd National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1999, CDROM distribution.

[37] "PKI Account Authority Digital Signature Infrastructure", Anne Wheeler and Lynn Wheeler, `draft-wheeler-ipki-aads-01.txt`, 16 November 1998.

[38] "Account-Based Secure Payment Objects", ANSI X9.59 draft, 28 September 1999.

[39] "Online Certificate Status Protocol — OCSP", RFC 2560, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle Adams, June 1999.

[40] "Simple Certificate Validation Protocol (SCVP)", Ambarish Malpani and Paul Hoffman, `draft-ietf-pkix-scvp-03.txt`, 12 June 2000.

[41] "Data Validation and Certification Server Protocols", Carlisle Adams, Peter Sylvester, Michael Zolotarev, Robert Zuccherato, `draft-ietf-pkix-dcs-05.txt`, 14 July 2000.

[42] "Web-based Integrated CA services Protocol, ICAP", Mine Sakurai, Hiroaki Kikuchi, Hiroyuki Hattori, Yoshiki Sameshima, and Hitoshi Kumagai, `draft-sakurai-pkix-icap-01.txt`, 31 January 1999.

[43] "Real Time Certificate Status Protocol — RCSP", Ambarish Malpani, Carlisle Adams, Rich Ankney, and Slava Galperin, `draft-malpani-rcsp-00.txt`, March 1998.

[44] "Web based Certificate Access Protocol — WebCAP/1.0", Surendra Reddy, `draft-ietf-pkix-webcap-00.txt`, April 1998.

[45] "Open CRL Distribution Process (OpenCDP)", Phillip Hallam-Baker and Warwick Ford, `draft-ietf-pkix-ocdp-00.txt`, 21 April 1998.

[46] "Directory Supported Certificate Status Options", Alan Lloyd, `draft-ietf-pkix-dir-cert-stat-01.txt`, 24 August 1998.

[47] "Roadmap issues", Bob Jueneman, posting to the `ietf-pkix` mailing list, message-ID `s79348e7.059@prv-mail20.provo.novell.com`, 19 July 1999.

[48] "Certificate Revocation and Certificate Update", Moni Naor and Kobbi Nissim, *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[49] "Fast Checking of Individual Certificate Revocation on Small Systems", Selwyn Russell, *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC'99)*, December 1999, p.249.