Depender Graphs: A Method of Fault-Tolerant Certificate Distribution*

Rebecca N. Wright
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932 USA
rwright@research.att.com

Patrick D. Lincoln
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025 USA
lincoln@csl.sri.com

Jonathan K. Millen
SRI International
333 Ravenswood Ave
Menlo Park, CA 94025 USA
millen@csl.sri.com

Abstract

We consider scalable certificate revocation in a public-key infrastructure (PKI). We introduce depender graphs, a new class of graphs that support efficient and fault-tolerant revocation. Nodes of a depender graph are participants that agree to forward revocation information to other participants. Our depender graphs are k-redundant, so that revocations are provably guaranteed to be received by all non-failed participants even if up to $k-1$ participants have failed. We present a protocol for constructing k-redundant depender graphs that has two desirable properties. First, it is load-balanced, in that no participant need have too many dependers. Second, it is localized, in that it avoids the need for any participant to maintain the global state of the depender graph. We also give a localized protocol for restructuring the graph in the event of permanent failures.

Keywords: fault tolerance, public key infrastructures (PKI), revocation

1 Introduction

Public keys and their certificates eventually become invalid. Most certificates have an expiration date, but for various reasons a certificate may become invalid prior to the expiration date. For example, the secret key may have been lost or compromised. The subject's identifying information, which might include an e-mail address or employer, may have changed. The certificate might have been used to enable organizational privileges that have since been withdrawn by the employer. Under these circumstances, there should be some way to revoke the certificate.

*A preliminary version of this paper appeared in *Proceedings of the Seventh ACM Conference on Computer and Communications Security (CCS)*, November 2000.

1.1 Existing Approaches

Current proposed standards for revocation, as found in the X.509 directory framework [14], and the Internet draft standard Public Key Infrastructure [1], involve *certificate revocation lists* (CRLs) maintained on key servers, which act as repositories for certificates. To revoke a certificate, the subject or another responsible authority sends the key server a *revocation notice*, which is a signed message identifying the certificate to be revoked.

Upon receipt of a valid revocation notice, the key server updates its CRL and no longer gives out the revoked certificate. In a push-based system, signed CRL updates are sent out periodically to interested users. In a pull-based system, end users who want to check the validity of a certificate must query the key server, and in response receive all or part of the latest CRL. Good discussions of revocation technologies can be found in [4] and [11]. There are various strategies for reducing communication and storage costs while maintaining timeliness of revocation, such as Kocher's certificate revocation trees [8] and related advances [7, 12], and methods for reducing server load, as in [2, 9].

One can try to reduce the need for revocation by limiting certificates to brief expiration periods, but this increases server load because new certificates must be sent more frequently. Rivest [13] suggested a two-level staged expiration, but this more complex system still requires a "suicide bureau" to maintain revocations due to key compromise. McDaniel and Rubin [10] suggest that revocation will remain a necessary part of any PKI.

In practice, it is important to recognize the fact that many certificates are issued by individuals, perhaps using PGP, and distributed without the use of a key server [15]. Certificates and revocations might be posted on Web pages to publicize them, but these pages typically do not support key server responsibilities such as CRL maintenance or distribution.

1.2 A New Distributed Approach

In this paper, we propose a new method for handling distribution of revocations or certificate updates. Our method is a push-based method in which each certificate has a list of *dependers*. Revocations and updates for a certificate, when they occur, are to be sent to the certificate's dependers.

Having a set of dependers for each certificate narrows the burden of notification to the minimal set of interested parties, which is an advantage in a push-based system. However, a solution in which a single root entity sends revocation notices for a particular certificate to all the dependers for that certificate has several disadvantages. If the root entity is a key server with many certificates and many customers, it may be too costly to provide and distribute customized CRL's for each of its customers. On the other hand, if the root entity is an individual, it need only be responsible for sending notices regarding its own certificate, but even so may not have the resources to distribute them to a large list. For example, everyone with a copy of the PGP software has the certificate of its creator Phil Zimmerman, and he would not and could not put everyone on his depender list. Finally, it is not fault-tolerant. For example, if the network link connecting a depender to the root entity is crashed or slow, then the depender will not be able to receive revocation notices in a timely fashion.

In our system, rather than having a centralized revocation server who sends revocations to all end users either periodically or in response to queries, the dependers themselves will participate in distributing revocations and other updates. To that end, participants who wish to receive revocation notices for a particular certificate must register as dependers on

"parent" participants. The participants can then be considered to form a *depender graph*. A participant agrees to forward any revocations or other updates she receives to her dependers. The source of a revocation notice sends the notice to dependers registered directly with it; those dependers then forward the revocations to their dependers, and so on.

The simplest kind of depender graph is a tree. For example, we could make a rule saying that anyone who relays a certificate should put the recipient on a depender list. That is, if A sends a certificate to B, then B would request to be on A's depender list for that certificate and A would agree to the request, regardless of whose certificate it is or where it came from. However, this simple scheme has the difficulty that it depends on the correct and prompt operation of participants, and that a participant who distributes a certificate to many users will also be bound to distribute revocations to them. Furthermore, it is even more vulnerable to failures than the centralized root entity scheme since there is generally only one path by which a revocation notice can be forwarded and the failure of any node on that path prevents all later nodes from receiving revocations.

In order to provide tolerance of up to $k-1$ crashed, slow, or misbehaving participants (or the network links connecting them), we require participants to register as dependers with at least k other participants. This straightforward idea, which we will elaborate on in the remainder of the paper, has several desirable properties:

- It is workable for individuals.
- It is "server-light," so that massive institutional facilities are not required.
- It is decentralized.
- It is survivable in the event of typical computer and network failures.
- It supports prompt revocation, even if some (up to k) components exhibit extraordinary delays.
- It requires only a realistic workload for those using the system.
- The workload is allocated in proportion to the self-interest of users.
- It makes it practical to distribute revocation information immediately, rather than delaying for a periodic CRL publication schedule.

Although we focus on using depender graphs to distribute revocations, they can also be used to distribute frequent short-lived certificates or other kinds of certificate updates.

In order to join a depender graph, a participant needs to find k other participants to depend on. We present joining protocols that are *load-balanced*, in that no participant need have too many dependers, and *localized*, in that no global state is maintained and participants need only maintain information about a few other participants. We also give a localized protocol for restructuring the graph in the event of permanent failures.

We define depender graphs and prove their fault tolerance properties in Section 2. We present depender graph construction protocols in Section 3. In Section 4, we present protocols to reconfigure the graph around permanent failures. We discuss some additional aspects of depender graphs in Section 5 and conclude in Section 6.

2 Depender Graphs

For a given certificate, we view certificate-holding participants in a network as nodes in a directed graph, called a *depender graph*, where there is an edge from A to B if B is on A's *depender list* for that certificate. In that case we say that B *depends* on A, and that A is a *parent* of B. We will always construct depender graphs to be acyclic and rooted graphs, and we say B is *below* A in a depender graph if there is a path from A to B. The root of the depender graph—usually the certificate subject or some kind of certificate server—is the source of revocation or update information about the certificate. When the root initiates a certificate revocation or update notice, it sends the notice to its dependers, called *root-dependers*. In turn, each node receiving the notice forwards it to its dependers.

In general, different certificates will have different depender graphs, though these graphs may share some common subgraphs. In practice, multiple depender graphs might have significant overlap, and some operations on them could be combined for efficiency. We do not discuss such optimizations further in this paper.

In order to avoid spurious revocations, revocation notices are typically authenticated by means of a digital signature. Since revocations are discarded if the authentication of the signature fails for any reason, malicious or arbitrary failures have the same effect as crash or omission failures, in which messages are lost. In order for a node to obtain the proper revocation information, it is sufficient that it receive one copy of the signed notice, regardless of whether other copies have correct information, incorrect information, or have been lost.

In our setting, the simplest method for signing revocation notices is that revocation notices of an individual's public key are signed by the corresponding private key; forwarded revocation notices maintain the initial signature. An advantage of this method is that since the key used to verify the revocation notice is the same as the key that is being revoked, a user will always be able to check the signature on revocation notices for certificates she has. Furthermore, a correct signature implies that the revocation notice either came from the owner of the private key and should therefore be trusted, or the revocation notice came from someone else who knows the private key or knows how to forge its signatures, in which case the key is by definition compromised and should be revoked.

If a public key is being revoked because the private key has been lost, or if the key is being revoked by an authority other than the owner of the private key, then it is not possible for the private key to sign the revocation. In this case, one possibility is for the user to first obtain a new set of keys and then use these to authenticate the revocation message, but this has the disadvantage of requiring the new key to be disseminated before the old key can be revoked. In the case that another authority is intended to be able to revoke the key, it is important that the dependers know the public key of that authority. This can be guaranteed, for example, by including the revoking authority's public key as an extension in the user's certificate. This is not necessary in the case that the revoking authority is the same as the certificate authority that originally signed the certificate.

We would like depender graphs to be fault-tolerant. Obviously, we cannot expect information from the root to be sent if the root has failed. However, the temporary or permanent failure of fewer than k non-root nodes should not prevent a revocation notice sent by the root from reaching any non-failed certificate holder in a timely fashion. Since revocations are digitally signed by a key known to all the dependers, it suffices to guarantee that each depender will receive at least one correct revocation notice in a timely fashion, independent of the timing or correctness of any other copies received. To guarantee that at least one correct revocation notice is received quickly, we consider the following k-redundancy property:

a rooted directed acyclic graph is *k-redundant* if even after the removal of any set of $k-1$ non-root nodes, there is a path from the root to every remaining node. (In Section 5.2, we address methods for making the root itself fault-tolerant if desired.)

We show below that the global property of k-redundancy can be achieved by ensuring a local property—that every node except for the root and its dependers has k parents in the graph; this is called the *k-parent* property. We refer to a rooted, directed, acyclic graph with the k-parent property as a *k-rdag*.

As mentioned above, since revocations are signed, we can essentially ignore malicious faults. However, if desired for other applications, depender graphs could be extended to tolerate malicious behavior during the distribution of information: i.e. non-root-depender nodes in a $(2k+1)$-rdag can tolerate up to k Byzantine failures using voting.

In order to prove the fault tolerance properties of k-rdags, we need some basic graph theoretic definitions, slightly modified to take into account the rooted nature of our depender graphs. A set of nodes is *root-avoiding* if it does not contain the root. A *cut set* is a root-avoiding set of nodes whose removal disconnects some remaining node from the root. Two or more paths from A to B are *pairwise internally node-disjoint* if no two of the paths have any nodes in common except A and B. In any rooted, finite, acyclic graph, it is possible to define a *rank* function on nodes such that every edge goes to a node of greater rank than the one it is from (so edges are rank-increasing). For example, the rank of a node can be the length of the longest path from the root to that node.

Theorem 1 *Let G be a k-rdag. Then G is k-redundant.*

Proof: Let G be a k-rdag and let C be a cut set of G. Note that if every cut set contains at least k nodes, then any set of $k-1$ or fewer non-root nodes is not a cut set, so all remaining nodes are connected to the root, and the graph is k-redundant. Hence, it suffices to show that C has at least k nodes. Let x be a node that is disconnected from the root in $G-C$, and define the *neighborhood* of x to be the set of nodes y on paths from the root to x in G such that no path from y to x has a node in C. These are the nodes between C and x.

Note that since C disconnects x from the root, x is not the root or a root-depender, and therefore x has k parents by assumption. If the neighborhood of x is empty, then every parent of x must be in C, and hence C has at least k nodes, and we are done. Otherwise, find a node y in the neighborhood of x of minimum rank. By the definition of a neighborhood, y also is not the root or a root-depender. Hence, by the k-parent property, y has k parents. Those parents must all be in C, for one that is not would be in the neighborhood of x and have rank less than y, a contradiction. Thus, C has at least k nodes, completing the proof. □

The following more explicit result will be helpful when we consider the efficiency of revocation distribution.

Theorem 2 *Every k-rdag has k pairwise interior node-disjoint paths from the root to any node.*

Proof: Let G be a k-rdag. If the root and a root-depender are both active, then there is always a path between them (consisting of the single edge that connects them). Suppose x is a not the root of G, and is not a root-depender in G. Then by the argument in the proof of Theorem 1, it follows that any cut set that disconnects x from the root is of size at least k. By Menger's Theorem (cf. [6]), it further follows that there are k pairwise interior node-disjoint paths from the root to x. □

3 Depender Graph Construction

Depender graphs grow as new nodes join the graph. We envision that a new node will join the graph for a particular certificate when it receives the certificate from one of the nodes already in the graph. In order to maintain the k-parent property, the joining node must either depend on the root or find k nodes to depend on that are already in the graph.

3.1 Necessary and Sufficient Conditions

We first address the conditions necessary to ensure that there are always enough available parents without overloading participants with too many dependers. Hence, a restriction on the choice of parents is a participating node is allowed to place a limit on its *number of depender slots*, that is, the maximum number of dependers the node is willing to support. It is clear that if nodes are not willing to have enough depender slots, then it will not always be possible to add new nodes to the graph, since once the root's depender slots are full, each new node requires k parents, each of which has an available depender slot, in order to join the graph. (A node has a depender slot *available* if the number of dependers that it currently has is less than its maximum number.)

We will show that it is enough for each new node to have k depender slots. First, we define a *kernel* as k nodes such that if the nodes are ordered, from largest to smallest, by the number of available depender slots they have, and if d_i is the number of available depender slots the ith node on this ordered list has, then $d_i \geq i$. That is, the nodes have at least k, ..., 2, 1 slots available, respectively.

Theorem 3 *A k-rdag can be constructed from any number of nodes that each have k depender slots.*

Proof: Begin with the root and make the next k nodes root-dependers. Subsequent nodes need to find k parents. We claim that when a kernel exists, another node with k depender slots can always be added to the graph, and there will still be a kernel; that is, the existence of a kernel is an invariant.

Note first that just after the k root-dependers are added, each of the k root-dependers still has all its k slots available, more than satisfying the requirement for a kernel. (In fact, the root-dependers form a kernel even if the ith root-depender has only i slots.)

For the proof of invariance, assume that a kernel exists. We can add a new node and give it k parents by taking one parent from each of the kernel nodes. This preserves the existence of a kernel, since the original kernel nodes now have at least 0, 1, ..., $k-1$ slots available and the new node can be added to the kernel with its k available slots. □

The kernel-based algorithm for adding nodes to a depender graph used in the proof above is called a *triangular* scheme. The result of adding eight nodes to a root using such a scheme is illustrated in Figure 1 for $k = 3$. To emphasize the regular construction of the graph, the root-dependers are shown with additional root-depender parents, though those edges are not necessary.

Note that a kernel may not be unique, and there may exist other nodes with additional available slots, because some nodes, such as those designed to be key servers, may support more than the minimum assumed k dependers for each certificate.

The triangular scheme always has $1 + 2 + ... + k = (k^2 + k)/2$ slots available once all the root-dependers have been added. This may sound excessive, since adding a node only

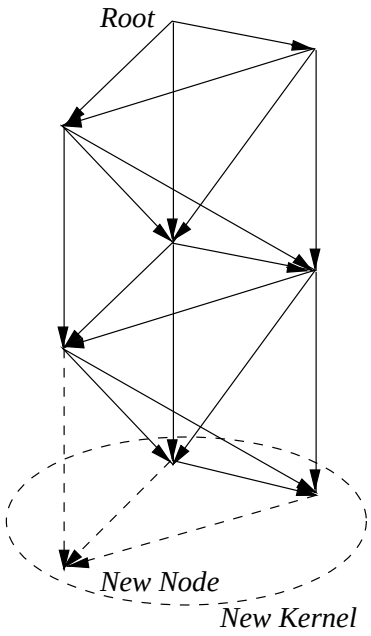

Figure 1: The $k = 3$ Triangular Scheme

requires finding k slots (in different parents), but we can show that this number $(k^2 + k)/2$ is minimal.

Theorem 4 *In order to add k non-root-depender nodes, a k-rdag must have at least $(k^2 + k)/2$ slots available.*

Proof: Consider adding a new set S of k nodes. The first node in S to be added must depend on k other nodes. So there must be at least one slot open in k other nodes at the beginning of the process of adding the S nodes. By the end of adding all nodes in S, a total of k^2 slots have been used. Each of the k additions needs to depend on k nodes, some of which may be in S. The maximum number of slots that may be used in the set S (with members of S depending on earlier members of S) is $(k^2 - k)/2$. Since k^2 total slots are used in adding S, that means there must have been at least $k^2 - (k^2 - k)/2 = (k^2 + k)/2$ slots at the beginning of the process of adding S. □

Hence, the triangular scheme is optimal in the sense of having the fewest sustainable number of available slots. Note that there may be other ways of achieving the same optimal number of available slots if some nodes are willing to support more than k dependers.

3.2 A Localized Protocol for Node Addition

One motivation of forwarding certificates and recording dependers for later revocation is that it is distributed and decentralized, so that it is not necessary for the root to communicate with all the nodes holding its certificate. Adding nodes with a triangular scheme seems to destroy this advantage by requiring participants to keep track of which nodes are in the current kernel. In this section, we show that it is not necessary to do so, because the

existence of a kernel can be maintained without knowing where it is, and we present a localized protocol that takes advantage of this.

Specifically, if there is a kernel and the parents of a new node are taken to be *any k nodes with available slots*, a kernel exists after the addition of the node. To see this, note that where kernel nodes are taken, an argument as in the proof of Theorem 3 shows that the new node plus all but one node from the old kernel form a new kernel. Where a non-kernel node is taken, the kernel node that "should" have been taken is still available to fill its role in the new kernel. Hence, the existence of a kernel is preserved. This flexibility in choosing parents makes it possible to consider optimization goals, such as minimizing the average path length in the depender graph.

Theorem 5 shows that if there is a kernel, then one can find k available depender slots in k distinct nodes by tracing down in the graph from any initial "search set" of k nodes.

Theorem 5 *If G is a k-rdag, then there is an available parent set below any set of k nodes.*

Proof: Let S be a set of k nodes in a k-rdag. Induct on the maximum length (i.e., the number of edges) of a path that begins in S and ends outside S. If the maximum is 0 then the S nodes have no dependers outside S, so each node in S can have at most $k-1$ dependers (all the other nodes in S), each node in S has at least one available slot, and thus S can be the parent set.

For the induction step, suppose the maximum such path length is n. If every node in S has an available slot, the k nodes in S can serve as parents. Otherwise, some node has no available slots, so it has a set S' of k dependers. The set S' has maximum path length smaller than n, and is below the original set S, so by induction and transitivity of "below" there exists an available parent set below the given k nodes. □

Theorem 5 suggests a localized protocol for adding new nodes, for which each node in the graph keeps track only of its parents and its dependers. Given a new node, we begin by identifying a single node already in the graph as a "starting node"; typically, the starting node would be a participant from whom a new participant has just learned a certificate. If the starting node does not have k parents, it must be the root or a root-depender. In that case, either the new node can be a root-depender, or if there are already k root-dependers, take those k nodes as a search set and apply Theorem 5. Otherwise, the starting node has k parents that can be taken as a search set as in Theorem 5.

It might be desired to choose parents in such a way that the path lengths from the root to each new node are minimized. The construction in Theorem 5 does not satisfy that property. To minimize path length, one would instead traverse back up the parent links and take depender slots from the highest available nodes. However, this would either require nodes to maintain more information about where in the graph the available slots are, or would require a new participant to traverse more of the graph in the worst case.

4 Reconfiguring After Failures

When a node wants to drop out of a depender graph, or is otherwise suspected to have failed permanently, we would like to be able to restructure the depender graph so that the k-redundancy property is maintained on the new graph. If such reconfigurations are done, the fault tolerance of our system over time can be much more than k, as long as there are not more than $k-1$ failed nodes between reconfigurations. In this section, we sketch a

protocol for reconfiguring the graph if only crash failures can occur. If malicious failures can occur, the reconfiguration protocol would need to be made robust in order to tolerate them.

Note that we need not replace failed nodes that have no dependers, since there are no other nodes affected by their failure. Similarly, since we assume that the root is the sole source of revocations, there is no use in replacing the root, since there will be no revocations sent as long as the root is failed.

In order to carry out a replacement, it is necessary that the local topological information stored in the failed node (its parent and depender addresses) has not been lost. To this end, we duplicate this information in one or more *caretaker* nodes. For the purposes of this exposition, we assume that only one failure may occur between reconfigurations. In our construction, we use one caretaker for each node that has one or more dependers. We call this node the *ward* of its caretaker. The actual replacement for the ward will not generally be the caretaker but rather some other node that does not have dependers at the time the replacement needs to be carried out. The replacement protocol described below will find an appropriate replacement node. Once the replacement node is identified, the necessary information is sent to the replacement from the caretaker and the replacement takes on the dependers and any caretaking duties of the ward being replaced.

Our protocol works if every node with dependers has a caretaker and a node can be the designated caretaker of at most one ward. This is initially true when a depender graph consists only of the root. The join protocol described below in Section 4.1 ensures that caretakers are identified as part of the procedure for adding new nodes, and does so in such a way as to preserve these properties.

There are a number of ways that failures could be detected. Parents could discover or suspect failures of their dependers through an acknowledgement requirement in the revocation forwarding protocol. Since revocations may only be infrequent, it may be desirable to detect and repair failures before revocations are to be sent. In this case, this could be achieved by requiring "I'm awake" messages to be sent from wards to their caretakers. Of course, such messages should not be sent too frequently, or their cost will outweigh the benefits of using depender graphs. If a node is replaced when it has not actually failed, but just suffered an unusually long delay, it can be reconnected when it recovers by treating it as a new node. An additional consideration in this case is that it may possess certificates that it believes to be valid, but which were revoked during the period in which it was disconnected. To avoid this problem, nodes should save revocations they receive until the affected certificate has expired; these revocations will be communicated to a recovering node as part of the recovery procedure.

If instead of our above assumption that only one permanent failure occurs between reconfigurations, we assume that ℓ permanent failures may occur between reconfigurations, it would be necessary for each node to store ℓ levels of topological information.

4.1 Sustainability of Join

Our reconfiguration protocol requires us to use a new join protocol for adding new nodes to the depender graph. A depender graph is *protected* if every parent has a caretaker, and some additional properties, stated below, are satisfied. The join protocol preserves the protected status. The key idea is that if a node becomes a parent, it is connected to the end of a chain of caretaker-ward pairs.

We say a node is *free* if it has no dependers (i.e., it is not a parent). A node is *available*

if it can accept another depender; otherwise it is *full*. A set of nodes is available if every node in the set is available.

As mentioned previously, a free node needs no caretaker as it has no dependers and will not be replaced. Also, the root needs no caretaker because the root is the source of all revocations, and so if it is not active, no revocations will be initiated. If root authority is distributed, as discussed in Section 5.2, we would have to modify the caretaker scheme accordingly.

Formally, we say a depender graph is *protected* if

1. every non-root parent has a caretaker,
2. every caretaker has exactly one ward,
3. every caretaker is a parent, and
4. there are no cycles in the graph of caretaker-to-ward edges.

Part of the reason to avoid caretaker-ward cycles is to increase the number of simultaneous failures that can be recovered from. An n-cycle can recover from at most $n-1$ failures.

Theorem 6 *Suppose we are given a new node to be added to a protected depender graph, and set of k available nodes. Then it is possible to add parent responsibilities and caretaker responsibilities, if necessary, in such a way that the resulting graph is protected.*

Proof: Consider the caretaker needs of the nodes in the available set A. Tentatively add depender edges from each node in A to the new node; we may later change one of these edges. Nodes that were not free (i.e., already parents) prior to adding the new node must already have caretakers.

If some nodes in the available set were free, say $A' \subseteq A$, we must find caretakers for these nodes. Choose a node in A', and consider one of its parents, p. If p is already a caretaker, follow the caretaker-ward chain from p to its non-caretaker terminus q (which has a caretaker because we followed the path through its caretaker to arrive at q); otherwise let $q = p$ (which has a caretaker unless it is the root, because it is a parent). Thus, we have found a non-caretaker q that itself has a caretaker unless it is the root.

Before we make q a caretaker, we have to make sure q is a parent. If it is not, we replace some member r of A' with q. That is, we replace the depender edge from r to the new node with a depender edge from q, leaving r as a free node. We assign $A' := (A' \setminus \{r\}) \cup \{q\}$, and now proceed to assign caretakers for all nodes in A'.

To do this, we extend the caretaker-ward chain from q through all nodes in A'. We can do this because all nodes in A' were not parents and therefore were not already caretakers. Since all nodes in A now have the new node as a child, all conditions for being protected are satisfied. □

Note that the above construction actually results in a single caretaker-ward chain starting at the root and traversing through all the parent nodes of the graph.

Next we show how available sets are located. When the depender graph is first created, it has a root. New nodes are added as root-dependers, requiring no concern about caretakers or k-redundancy, until the root is full. Once the root is full, a new node is added to a depender graph by asking some prior node to find k parents for the new one. The parents of

the prior node serve as the first candidate set. The nodes in this set are either all available or not. If so (skip this case if the prior node is a root-depender), they become the parents of the new node. If not, there is a full parent, and k of its children become the next candidate set. This iteration terminates because the graph is finite and acyclic, as illustrated in Figure 2. Caretaker responsibilities are assigned in accordance with Theorem 6.

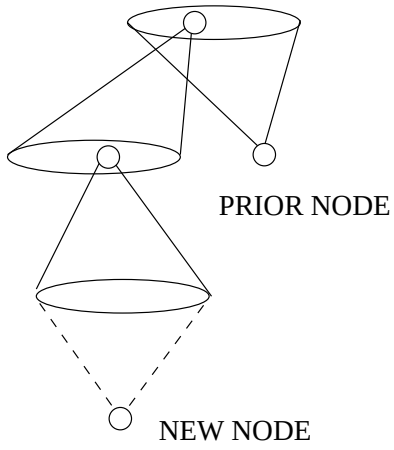

Figure 2: Searching for Parents

4.2 The Replace Protocol

When a caretaker node decides that its ward must be replaced, it must locate a free node to serve as a replacement. To do so, it sends a "help" message to one or more of its dependers, containing its address. The depender passes the message down again until it reaches a free node. The free node replies to the caretaker node, and the caretaker node selects a replacement from among the replies (if there are more than one). The replacement receives a copy of the ward's backup information from the ward. The replacement becomes the new ward of the caretaker.

As with the initial selection of parents from different available sets of parents, the selection of which depender should pass down the help request as well as the selection among potential replacements that reply to the caretaker, could be made arbitrarily or could be made according to some strategy for selecting the best candidate.

The updated free node, the old ward's replacement, tells the ward's parents (whose identities it now knows) to replace the old ward by itself as a depender. The new ward accepts the old ward's dependers as its own without having to tell them, but if the old ward was the caretaker of another node, the new ward must contact that node to obtain a backup copy of its stored information.

There is a problem if the the old ward had more than k dependers, since new nodes are not required to support more than k dependers. To handle this, we assume that the help request specifies the number of dependers. If a willing replacement is found, it can be used, otherwise the replacement protocol fails. This may be expected to happen if the failed node is a server with a very large number of dependers. It is not unreasonable to expect that such nodes are well maintained, and are unlikely to fail permanently without some administrative solution in the event of failure.

5 Discussion

In this section, we briefly discuss a number of issues and possible extensions where further research is called for.

5.1 Link/Transport connectivity

If two paths are node-disjoint, then they are also edge-disjoint. Thus, our depender graphs are tolerant against the failure of $k-1$ node or edge failures. However, in a real network, links between different nodes are not independent. Often many links go through the same switching node in an underlying communication infrastructure. Thus, the failure of one switching node may result in the failure of many edges in a depender graph.

It would therefore be desirable to assure that links from a node to its k parents are independent (so it takes k failures of lower-layer switching nodes to break them all). If in addition there are k independent paths from the root to its dependers, then an inductive argument shows that it takes k failures of the underlying network components to cut all paths to a node. A weaker version guarantees k-redundancy for non-root-depender nodes so long as each link from the root to a root-depender is independent of all other links in the graph. We can still show by induction that it takes k failures to cut off a non-root-depender.

Checking independence of transport paths can be done using network monitoring tools such as "traceroute." However, in practice this information is rather dynamic and may be difficult to keep a handle on.

5.2 Distributing root authority

In some settings, it is desirable for the root authority to be distributed among multiple parties, so it takes the participation of at least t of these parties to send out a valid revocation notice (and furthermore *any* t parties can do so). This can be achieved by distributing the functionality of the root into multiple parties and using threshold signatures (c.f. [3, 5]) so that the correct participation of t parties is necessary and sufficient to create a valid revocation. If this new "distributed root" consists of at least $k+t-1$ parties, then this also provides crash fault tolerance for up to $k-1$ of the root parties.

In order for the threshold signatures to work, the root-dependers must now have at least $k+t-1$ of the root parties as parents; other nodes still need k parents as before. When a revocation notice is sent, it is signed using the threshold signature scheme. Each root node sends its partial signature to the root-dependers. The root-dependers reconstruct the signed revocation notice, and if it is a valid signature, they proceed as before by forwarding the signed revocation. Since the resulting depender graph has its normal properties with respect to this distributed root, it still enjoys the k-redundancy property with respect to it.

5.3 Reusing Links

In some applications it may be desirable to reuse all or part of the depender graph to support all-to-all revocation-like signaling. As described above, k-rdags must be constructed for each root node wishing to have a reliable revocation mechanism. However, one can take great advantage of sharing between multiple k-rdags to reduce resource usage in large communities. Further, if all links can be assumed to be used in both directions, one could build a single k-rdag that supports reliable revocations or other communication from any

node in the network using a simple flooding protocol. However, the details of this approach are beyond the scope of this paper.

5.4 Global Optimizations

For distribution of revocation notices, the k-redundancy property can be exploited simply by having each node forward a notice to all its dependers. In the general case, this is the best that can be done. However, there are several situations in which global information about the graph could be used to reduce or eliminate unnecessary network traffic while still ensuring revocations are distributed properly.

For example, if the graph has more than k disjoint paths to some nodes, it might be possible to remove or ignore some of the edges of the graph. Similarly, if not all nodes need to receive each update, then some edges can be removed. Given a particular destination node, Theorem 2 says that there are k pairwise interior node-disjoint paths from the root to that node, so that using only the edges in these paths would eliminate unnecessary traffic while preserving k-redundancy with respect to that one destination node. When only some subset of nodes needs to receive a revocation notice, the goal would be to find a minimal set of edges that include k disjoint paths to each node in the subset. Finally, in the case that something more is known about which failure configurations can occur than just that any $k-1$ nodes might simultaneously fail, it might be possible to ensure that each node has always at least one path from the root through no failed nodes without having k disjoint paths to each node.

6 Conclusions

Depender graphs provide a locally manageable, scalable, efficient, and fault-tolerant method of certificate revocation in a public-key infrastructure. The relationship between k-failure protection and the obligation of a participant to support k dependers meets the objective of having a fair and realistic workload, and we have shown how the system responds to both temporary and permanent failures. Many practical issues remain open, such as how k should be chosen to balance fault tolerance needs with efficiency considerations.

Due to their fault tolerance and localized construction protocols, k-rdags may find useful applications elsewhere. As described in this paper, they are most useful for environments in which only crash or delay failures occur, or if the information to be sent is digitally signed or otherwise verifiable, as in our case of certificate revocations. However, depender graphs can be extended to tolerate malicious behavior during the distribution of information: i.e. non-root-depender nodes in a $(2k+1)$-rdag can tolerate up to k Byzantine failures using voting; additionally, the root-dependers must have some means for verifying information received from the root, such as voting among themselves.

Other possible applications that could possibly benefit from depender graphs include fault-tolerant multicast backbone (MBone) trees, distributing routing information in the Internet such as reachability information exchanged by the BGP protocol, and maintaining location information for a mobile host as it moves from one base station to another.

Acknowledgments

We thank Andrea Lincoln and Patrick McDaniel for helpful discussions.

References

[1] C. Adams and R. Zuccherato, "Internet X.509 Public Key Infrastructure Data Certification Server Protocols," Internet Draft, PKIX Working Group, 1998.

[2] D. Cooper, "A Model of Certificate Revocation," *Proc. 15th Annual Computer Security Applications Conference,* 1999, 256–264.

[3] Y. Desmedt and Y. Frankel, "Shared generation of authenticators and signatures," In *Advances in Cryptology—CRYPTO '91*, *Lecture Notes in Computer Science* 576, 457–469, Springer-Verlag, 1992.

[4] B. Fox and B. LaMacchia, "Certificate Revocation: Mechanics and Meaning," *Proc. Financial Cryptography '98*, LNCS 1465, 1998, 158–164.

[5] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust Threshold DSS Signatures," In *Advances in Cryptology—CRYPTO '96*, *Lecture Notes in Computer Science* 1070, 354–371, Springer-Verlag, 1996.

[6] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[7] H. Kikuchi, K. Abe, and S. Nakanishi, "Performance Evaluation of Certificate Revocation Using k-Valued Hash Tree," *Proc. ISW'99*, LNCS 1729, 1999, 103–117.

[8] P. Kocher, "On Certificate Revocation and Validation," *Proc. Financial Cryptography '98*, LNCS 1465, 1998, 172–177.

[9] P. McDaniel and S. Jamin, "Windowed Certificate Revocation," *Proc. IEEE Infocom 2000*, IEEE, 2000, 1406–1414.

[10] P. McDaniel and A. Rubin, "A Response to 'Can We Eliminate Certificate Revocation Lists?' ", *Proc. Financial Cryptography 2000*, February 2000.

[11] M. Myers, "Revocation: Options and Challenges," *Proc. Financial Cryptography '98*, LNCS 1465, 1998, 165-171.

[12] M. Naor and K. Nissim, "Certificate Revocation and Certificate Update," *Proc. 7th USENIX Security Symposium*, 1998, 217–228.

[13] R. Rivest, "Can we eliminate certificate revocation lists?" *Proc. Financial Cryptography '98*, LNCS 1465, 1998, 178-183.

[14] "The Directory-Authentication Framework," CCITT Recommendation X.509.

[15] P. Zimmermann, *The Official PGP User's Guide,* MIT Press, 1995.