EKA: Efficient Keyserver using ALMI

Marcel Waldvogel Radhesh Mohandas Sherlia Shi
Applied Research Laboratory
Washington University in St.Louis
{*mwa,rod,sherlia*}*@arl.wustl.edu*

Abstract

The keyserver network serves as a repository of OpenPGP keys, providing replication throughout the Internet. It currently uses an inefficient and insufficient protocol to keep its nodes synchronized: highly redundant network traffic and excessive overhead due to several thousand e-mail messages per day. Under these conditions, even short network outages cause massive mail server overloads and losses, resulting in continuously diverging databases. In this paper, we present a new protocol to achieve complete synchronization efficiently and automatically, drastically reducing the need for manual intervention. Our protocol transmits only the updates and uses multicast to optimize the amount of data sent. Since support for native multicast is not widely available in the underlying network and current Internet multicast does not scale well, we base our keyserver on ALMI. ALMI *is a middleware for reliable application-level multicast, providing scalable join/leave notification of neighbors, significantly reducing the complexity of the application. As a part of this work, we have also implemented a keyserver software which uses our protocol and an efficient* RDBMS[1] *[Ora] back-end to hold the keys.*

Keywords: PGP, keyserver, ALMI, synchronization, add mostly database, high availability

1 Introduction

The *OpenPGP* keyserver network serves as a repository of *OpenPGP* keys, providing replication throughout the entire Internet. Replication is provided as a means to keep the keyserver infrastructure and the stored *OpenPGP* public keys available independent of attacks, be they through denial-of-service or through political or legal systems. Currently, there are some 30 servers worldwide as replica sites under independent, autonomous administration. Unfortunately, the distribution mechanism currently employed uses a node synchronization protocol that was not designed for the load experienced now and expected in the near future. Redundant traffic is sent through the network in the form of e-mail messages, resulting in a huge data exchange overhead through many thousands of e-mail messages received daily at a single server. Even on small network or server outages, this results in mail server overloads. Despite the redundant traffic, these outages result in large-scale database divergence. In this paper, we present a new protocol to address these issues while maintain the existing positive aspects, namely, independent operation of all databases, i.e., the absence of vulnerable "master" databases, and complete replication (not just distribution). Although there is no single master node, the protocol provides for "caching" nodes, which only keep a small subset of keys, generally only those interesting to the operator of this node. Such "slave" nodes will still receive important updates, such as of key revocation certificates [CDFT98, Fei00], immediately. It also allows for the efficient operation of databases which are not continuously connected to the Internet.

Besides assuring the successful copying of the data to all participating servers, the distributedness of the replication process introduces concurrent updates, potentially to the same key. This concurrency is amplified by the desire to deal gracefully with extended periods of outages and the potential for offline operation. The synchronization mechanism currently deployed among the *OpenPGP* keyservers addresses the concurrency problem by having each keyserver performing the necessary merging operations independently, leading to possibly different results. Please note that although these local decisions can cause differences, the vast majority of the actual divergences among the current keyserver databases is not due to these independent decisions, but due to the loss of replication updates. The proposed new scheme obtains a globally consistent database by deferring the potentially diverging decisions to the client importing the key. This delegation also results in a trustworthy crypto system without requiring extensive amounts

[1] Relational Database Management System

of trust of the user into the key server infrastructure.

The current protocol is also inefficient as the entire key (some of them weigh in at several tens of kilobytes) is transmitted when it is altered, not just the changes. The goal of this work is to replace the existing scheme by an efficient protocol that transmits only the updates by reliable multicast and uses an efficient RDBMS back-end to hold the keys. The scheme developed here can easily be generalized to any add mostly database. By add mostly database we refer to database schemes where the number of deletions and modifications is far less compared to additions and to schemes where removing the entries in the database is not an option.[2] Examples of such databases are customer service databases, mail archives, problem tracking utilities, software distribution updates, distributed document annotation tools, applications for Computer Supported Collaborative Work (CSCW) and version tracking mechanisms. These databases are different from simple mirrors because updates occur at any/all of the nodes and have to be replicated across the network to the peers.

Later in this section, a brief introduction of various terms and concepts used in our work is given. In section 2 we present the existing keyservers. We start with the problem of synchronization and explain our scheme in section 3. We explain in detail the role of the middleware package for multicast in section 4. We present our evaluations and comparison in section 5 and conclude the results in 6

1.1 Background

Pretty Good Privacy (PGP) [Sta94], a program written by Phil Zimmerman takes the credit of being the first widely available and accepted cryptographic software that supported the Public Key Infrastructure. A family of open source cryptographic software use PGP keys which provides the necessary framework to store different types of asymmetric cryptographic keys in a well defined format. RFC 2440 [CDFT98] governs the definition of the *OpenPGP* message format.

The security model that PGP believes in is called the Web of Trust [Fei00]: Each user can certify the keys of any set of other user. A signature is trusted if there is a trusted path of signatures between the signing key and the trusted key. The advantage of this model is that there is no centralized certification authority (CA) into which everyone has to have ultimate trust. The Web of Trust generalizes on the CA model by making everyone a potential (and potentially trusted) CA.

A keyserver is a passive repository of keys which allows users to submit and retrieve keys. A group of redundant servers have formed a worldwide forum [Key] where the Internet community can publish and retrieve its public keys. In this way no single keyserver has to be completely trusted and the community can exchange keys with its nearest keyserver and the model continues to work even if some keyservers are down. These keyservers have become very popular. The number of keys submitted to and available at the keyservers is steadily increasing and doubling every six months. Each of these keyservers contains the complete set of public keys submitted to the forum. Hence the PGP key database forms a completely replicated database which is an extreme form of distributed database.

The *Lightweight Directory Access Protocol* (LDAP) [WHK97] was defined for the quick implementation of the X.500 series Directory Access Protocol on the TCP/IP stack. It is very efficient to store properties on hierarchical names using LDAP. LDAP allows for the creation of a Public Key Infrastructure(PKI) for secure exchange of information over an insecure network and is used by some keyservers.

2 Current keyservers

The first keyserver was a wrapper to the PGP program and consisted of an email and a web interface. As the number of keys submitted to the keyservers grew to several thousands, the need for a more efficient mechanism to store the keys in a database was felt. Hence Marc Horowitz wrote *pksd* [Hor97], an *OpenPGP* keyserver software, which became the de-facto standard for the public keyservers. Besides this one, Network Associates (NAI, the current owners of PGP) supplies a keyserver called *pgpcertd* using LDAP. Table 1 compares ours to these keyservers.

Keyserver	**pksd**	**pgpcertd**	**EKA**
Database Diverging	yes	no	no
delayed updates	no	yes	no
synchronization means	email	CIP[AL97]	socket
uses multicast	no	no	yes
fundamental unit	key	key	packet
redundant transmissions	yes	yes	no
manual intervention	yes	yes	no
Open source	yes	no	yes

Table 1. Different Keyservers

pksd: It uses Berkeley DB2 embedded database [Ber] to store the keys. This keyserver stores PGP keys as objects in the database. The fine-grained packet structure [CDFT98] of the key is not used and whenever a single packet is added, the entire key is updated and transmitted. This software uses an email interface to keep the keyservers synchronized in a manually configured, dense mesh, resulting in a keyserver typically receiving the same update $O(k)$ times, where k

[2] For security reasons, modifications to a *OpenPGP* key are always done by adding new information.

is the degree of that node. To counteract the continual trend to diverge, the administrators try to synchronize their databases by exchanging entire dumps of the database and merging them into their corresponding local databases. This merging operation requires large amounts of administrator time, CPU time, and network bandwidth. With the current increasing rate of submitted and updated keys, it is to be expected that the email interface will not be able to keep up with the increments.

pgpcertd: Public keys that are certified by a Certification Authority (CA) are best stored and distributed using a LDAP keyserver. In this certificate based security model the CAs typically form a hierarchy with a "root" CAs which everyone has to trust completely. As this ultimate trust is not feasible, a large number of root CAs have developed. If the CA certifying (and thus storing) a key is not known (and thus trusted) by the user, she has to recursively scan the parent CAs until she reaches a trusted CA. The LDAP servers are configured according to the CA hierarchy and changes made by any CA are propagated accordingly to the entire keyserver network.

The NAI keyserver uses this feature of LDAP. Currently, these server does not provide for an automatic synchronization mechanism with the other *OpenPGP* keyservers. Also the LDAP based scheme benefits most when there is a hierarchy to exploit. So for a pure Web of Trust model, there is no hierarchy in the data that can be used by LDAP and there is no special advantage to use it as a distributed database scheme.

This keyserver also gives the users more control over their keys, which is hard to achieve in a distributed manner.

3 Synchronization

The updates made at any one database have to be propagated to the entire distributed environment. To make this task of global synchronization easier, most of the distributed database systems today have a single master database to which all the changes are applied before being sent out to the replicas. This schema is not acceptable to the keyserver network where anybody in the Internet can (and should be able to) host a full-fledged keyserver which can be updated by the users. Database replication with update-anywhere capability while maintaining global synchronization and isolation is considered as one of the hard problems in distributed databases. The current solutions implemented cost expensive resources or have a delay in reflecting the data at all sites. Distributed RDBMS packages are too general and do not take advantage of any structure in the data they maintain, or in the limited set of operations supported by that data.

In this section we first introduce the different synchronization mechanisms before presenting our scheme.

3.1 Replicated Database Synchronization

In a multi-master replication environment, all master sites communicate directly and continually propagate data and schema changes. The synchronization mechanism works to converge the data and to ensure global transaction and data integrity. In asynchronous replication, the updates are stored in a deferred transaction queue at the master site where the change occurred. These deferred transactions are then periodically propagated to other master sites. The disadvantage of this method is that the changes are not visible on all masters immediately and conflicts will arise due to simultaneous updates at different masters. In synchronous replication, changes made at one master site are propagated immediately to all participating master sites. To achieve this, a global transaction is implemented which succeeds iff the transactions on all masters are successful. The disadvantage of this method is that the global transaction does not complete if one of the master sites is unreachable or cannot serve the request for any reason. Also, it depends on *a priori* knowledge of all masters. It has been shown that this method degrades the performance of distributed replicated database system and the degradation increases with the number of nodes and the distances between them. A practical compromise between the two modes is to have *deferred transactions*. In the first phase the changes are committed at the local site and the transaction locks are released very fast. In the second phase the transaction is propagated globally on all the master sites after a delay. If it fails at one site, then the local transaction is rolled back. Since conflicts are very rare this approach is suitable and used in most Distributed Database packages. However a lot of information has to be stored for the two level rollback in this system and it is very complex to implement.

3.1.1 LDAP Synchronization

LDAP servers keep themselves synchronized by using an exchange protocol called Common Indexing Protocol (CIP) [AL97]. For synchronization among these servers there has to be a replication agreement in the distributed environment. This agreement is usually in the form of a manually maintained configuration file. All the servers in the environment are classified in one of the four categories: Master, Read-only slave, Read-write slave and No-replica node. LDAP uses a lazy replica based protocol for synchronization. To maintain consistency in the distributed environment, multi-master directory infrastructure need to be avoided. In multi-master networks there are different topologies in which the masters are connected. Each node queues the updates into a replication log file which is transmitted periodically [PGP]. The trade-off between redundant transmissions, consistency, and reliability inspires the decisions for selecting synchronization parameters including the network topology used.

3.1.2 EKA Synchronization

In add mostly databases, the frequent operation is the (idempotent) addition operation. There will not be any conflicts in the global transaction if the local transaction goes through, provided we take care of some identification issues. Hence we can provide a synchronization scheme that has the latency of asynchronous transmissions and consistency of synchronous transmissions. We release the transaction locks as soon as the local transaction completes. Each transaction results in an addition of an object and a globally unique serial number is assigned to the object at the local database. Since we are using reliable multicast a transaction transmitted over the network is assumed never to be lost as long as the receiving node is connected to the network. We have developed a protocol to synchronize the node if it reconnects to the network after a brief period of inactivity. Our mechanism need not block if one or more nodes do not receive the updates. This is important as some of the nodes may be disconnected from the network and when this happens, the network must continue to progress without any overload.

In some cases, it may be required to delete an object in the database. When this needs to be done, there is a potential for inconsistency when the object deleted at one node is updated at another node. To deal with this issue, we mark the object to be deleted with a flag and not actually remove it from the database. Hence such conflicts can be automatically resolved by performing the merge of the update operations which lead to the union of the changes. This also prevents loss of information in the consistency by serialization scheme. However since deletion is assumed to be a very rare operation, the space overhead of maintaining the deleted object in the database is not significant.

3.2 Enabling Technologies

3.2.1 Serial Numbers

For our scheme to work, we require each addition performed at a local node be assigned a globally unique identifier. We do this by assigning a unique serial number to each object added to the database. A serial number contains two parts: A host identifier (hostID) and an integer sequence number. The hostID is the IPv4 address of the keyserver which first sees the object. The sequence number is locally uniquely assigned by the owner of the hostID, the concatenation with the hostID ensures global uniqueness, allowing for the generation of unique identifiers in a distributed and potentially disconnected environment. By simply having the locally unique part a counter, these identifiers can also be used to quickly check which of two peers has newer information about a given hostID.

A PGP key is a partially ordered set of packets. A packet is the smallest unit that can be added to a key or modified. When this happens it suffices to transmit only the packet to supply all the update information. We associate this packet with the sequence number to distinguish its identity. The PGP keys are governed by a hierarchical arrangement of packets. So with each packet we also associate it's parent packet's serial number to enable reconstruction of the complete key during retrieval. Once a serial number has been associated with a PGP packet, it is called a *serial packet* which is the fundamental object stored in our keyserver database and reliably multicast to all other servers.

3.2.2 Multicast Transport

To reduce duplicate transmission of data over the network, we remove the redundant links by forming a spanning tree and using it for multicast. Our keyserver requires a reliable multicast mechanism which guarantees that a node will not miss any packets as long as it is connected to the network. Since the keyservers will be randomly scattered around the Internet and since their number will be relatively small (not more than a few hundred or thousand), application level multicast (Section 4) is probably the most suitable option. Baumer [Bau98] considered using the *MBone* [Eri94], but then all the keyservers may not be connected to the *MBone*. Further, the typical high loss rates associated with *MBone* would have made the reliable multicast connections difficult. Also, having well-defined group join and leave events significantly simplified design and implementation.

3.2.3 Startup synchronization protocol

The keyserver network has to continue working even when some of the nodes are disconnected from the network. When this happens, the multicast mechanism reconfigures itself to contain only the connected nodes and update propagation continues as normal [PSVW01]. Only the disconnected nodes will remain unsynchronized. When a node rejoins the network, it must receive all the updates that it missed during its down time. To achieve this each node registers the highest sequence number that each host has issued. Since the sequence numbers are issued in a strictly increasing order, they can be used as an extremely compact log of the updates that the node has successfully received and depicts the exact state of the node before disconnection. When the node reconnects, it makes an out-of-band connection to its new neighbor and sends its state before disconnection. The neighbor is synchronized with the rest

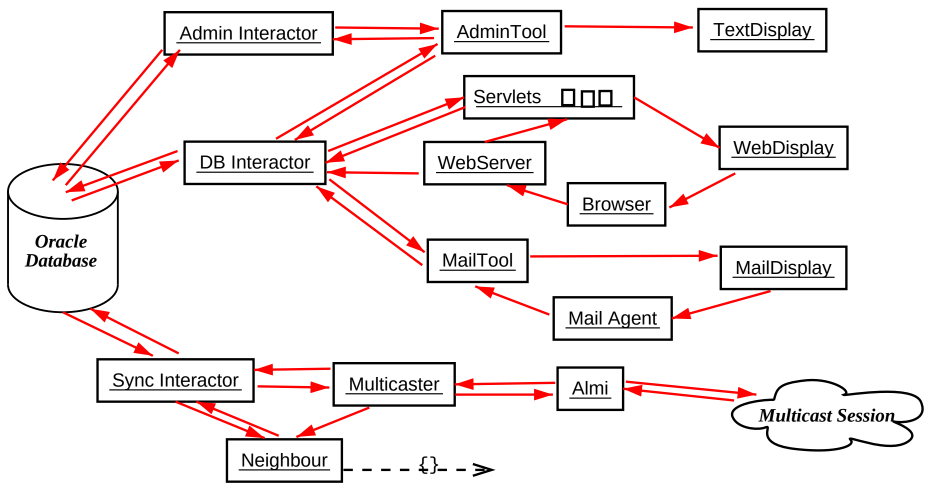

Figure 1. EKA **component interaction**

of the network and also has its latest state. It sends the missing updates to the reconnecting node using unicast, after retrieving them from the local database. In this way we do not cause any extra traffic or load on the remaining network. If the reconnecting node was still working and receiving updates from local users during the time it was disconnected, it can potentially have received updates and have its state incremented. In that case the network would not have received these updates and by requesting the neighbor's state, the node can reconstruct the updates and retransmit them using ALMI. The startup protocol that every node follows when it connects to the network will be as follows.

1. Send state to neighbor.
2. Receive missed updates from neighbor.
3. Receive state of neighbor (This is also the state of the keyserver network).
4. Regenerate the updates to the keyserver network from its advanced state by comparing with the neighbor's state.
5. Multicast these updates to the keyserver network using ALMI.

3.2.4 Deletion and Conflict Resolution

A key submitted to the keyserver network can not and should not be deleted. This is because one of the functions of keyservers would be to serve as archives for the keys that an user had been using at a previous point in time. If the user no longer wishes to use that key, then as per the PGP framework, he must submit a revocation signature declaring that the key is no longer valid and this will be one more packet that will be added to his key. So anybody can download his revoked key from the passive keyserver repository and it is up to the encryption software that he uses to deal with the revoked keys. The only conflict that can arise is when a packet is simultaneously submitted to more than one keyserver. This is not a commonly occurring situation and occurs only when a user deliberately sends updates to more than one keyserver. Under this circumstance the same packet will have more than one sequence number and this may eventually disturb the hierarchy of the PGP key. So when a keyserver receives an update and finds that the data packet is already present in its database, it retains the packet with the lowest serial number. Each node follows this policy consistently and eventually the conflict is resolved as each update reaches all the nodes connected to the network. Hence the hostID in the serial number becomes the tie breaker.[3]

3.2.5 Independent Tool Architecture

To enable the services of the EKA keyserver simultaneously available through different interfaces, the keyserver is implemented as a set of independent tools which can continue to work in advent of failure of others. In this way the keyserver remains to be partially available in advent of failures of some of the service agents. EKA requires a RDBMS with transaction support to store the PGP keys and assumes that this RDBMS is always running as long as the keyserver host is up. There is a synchronization tool that connects to the multicast session whenever the keyserver connects to the

[3]Alternatively, a set of sequence numbers could be used to identify a unique key.

network and sends and receives updates as describe in the previous sections. The keyserver can be accessed by web or mail by the Internet community. The administrator is provided with an AdminTool to do more privileged operations, collect statistics, perform data-processing operations and maintenance. The interaction between the various components that constitute the keyserver is depicted in figure 1

3.2.6 Event driven model

EKA is built on a event driven model. The different events that trigger of actions are network connection, disconnection, starting and termination of a tool, request or command from a user, addition or modification of a key. A user connects to the database using one of the interfaces and issues requests which are processed independently as far as the user is concerned and the synchronization operations are transparent to him. Whenever a database is altered it triggers of events that communicate with the synchronizing tool and multicast the updates. If the network is not accessible, the changes to the database are logged and propagated on reconnection.

4 ALMI Communication Channels

ALMI[PSVW01] is an application level group communication middleware, that is tailored toward support of multicast groups of relatively small size with many to many semantics. In contrast with IP multicast, it does not rely on network infrastructure support, and thus, allows accelerated development and experiments of multicast applications anywhere in the network. Due to its functionality as a reliable mutlicast transport and its immediate availability, we have integrated our key servers on top of ALMI. In this section, we describe briefly its architecture and functions, more interested readers are directed to reference [PSVW01].

An ALMI session consists of a session controller and multiple session members. Session controller is a program instance, located at a place that is well known to and easily accessible by all members. The session controller manifests itself only in the control path. It handles member registrations and maintains the multicast tree's connectivity and efficiency in the dynamic network environment. Session members are connected via a virtual multicast tree, i.e., a tree that consists of unicast connections between end hosts. The tree is formed as a minimum spanning tree (MST) where the cost of each link is an application-specific performance metric, in the current implementation, the round-trip delay between members. The multicast tree is a shared-tree amongst members with bi-directional links. Session members not only send and receive data on the multicast tree, they also forward data to their next hops along the tree. In order to preserve the efficiency of the multicast tree, a session member monitors performance of unicast paths to and from a subset of other session members. This is achieved by periodically sending probes to these members and measuring the roundtrip response delay. Delay measurements are then reported to the controller and serve as the costs used to calculate a Minimum Spanning Tree. To prevent service disruption, such tree reconstruction does not happen very frequently and other precautions have been taken to minimize possible data losses during a tree transition.

In ALMI, the leverage of existing reliable unicast transport, i.e. TCP, provides data reliability on a hop-by-hop basis, which implies that packet losses due to transient network congestion and transmission errors are eliminated. Instead, the main reason for packet losses in ALMI are due to multicast tree transitions, transient network link failures, or node failures. The resulting ALMI's loss characteristics are that packet losses are infrequent but usually happen in bulk. In order to provide a fully reliable tranport for applications such as EKA keyservers and to preserve application reliability semantics, ALMI implements a scheme that is capable of recovering packet losses during different time intervals and also includes an application naming interface for efficient data recovery. Upon loss detection, a *NACK* is sent in the direction back to source and is aggregated at each upstream hop. When applications can buffer data or regenerate data from disk, data retransmission can happen locally. In this case, the node above the lossy link will retransmit data to the requesting subtree. Otherwise, when upstream node has reset its data states and can no longer retransmit data locally, the requestor initiates an *out-of-band connection* directly to the source, and subsequent request and retransmit are conducted over this out-of-band connection. In the extreme case of a long network blackout, it is likely that application does not need to recover all intermediate packets but only those that contain the most recent changes to the objects. In this case, ALMI implements a callback function to notify application of such network changes and allows application to recover packets of their own choices through the naming interface. Additionally, ALMI also deploys *ACKs* to synchronize data reception states at members. This is necessary for applications that require total reliability but have limited buffer space. The frequency of the *ACK* process depends on both the data rate and the smallest buffer space at a member application.

5 Evaluation and Comparison

Our test network consisted of 5 *Linux* machines each running Oracle 8i RDBMS. Four of them were on a LAN and the fifth one was connected by a DSL link outside a firewall. The time to add the entire keyring to the server was around 12h. We measured that the keyservers receive around 8000 new packets every day and these are transmitted to a (re-)joining node in the matter of a few minutes. This means a keyserver that is disconnected from the rest of the network for a few weeks will automatically be synchronized completely with the rest of the network in just a few minutes. Our test server received incrementals from 3 other pksd servers and these contained around 48% repeated keys. Among the remaining, 46% were new keys. Among the modified keys the number of packets sent out was around 30% of the number of packets to transmit entire keys.

EKA **and pksd:** In a typical pksd setup, the keyserver that receives a new key passes it to sendmail which mails the keyring to all the sync-sites with huge mail headers. At each sync-site, the sendmail transport receives the mail, spawns a procmail process which spawns another process to send the key to the keyserver daemon. These overheads and loads on the servers are avoided in our keyserver as updates are sent over sockets instead of emails. No new processes are spawned upon the arrival of an incremental. The PGP data is sent out in the Binary Format instead of the Armored Format reducing the payload size to 3/4 of the original. The deployment of a RDBMS enables scaling and makes it possible to have more complex queries on the database than currently possible. The multicast group continues to function without any errors even if a couple of nodes fail (crash or are disconnected). With pksd, if a sync-site fails, there will be numerous mail bounces and manual intervention by administrators may be required. Complete synchronization can be achieved in EKA by detecting the discontinuities in the serial numbers.

EKA **and pgpcertd:** Even in the pgpcertd, the entire key needs to be transmitted. If the same key is altered at one place and deleted at another at the same time, then an infinite loop may form as the keyserver does not keep any state information. In EKA the network topology that is ideal is automatically arrived at in a manner which is completely transparent to the keyserver, no replication agreements are required to be maintained and manual intervention for collision resolution is avoided.

6 Conclusion

In this paper we have presented a new protocol to keep add mostly databases synchronized. A new scheme to implement global consistency by adding a global identifier to each object in the database has been proposed. We have developed a *OpenPGP* keyserver as a proof of concept. We have successfully deployed a ALMI middleware package and demonstrated a good example for an Application Level Multicast application. We have evaluated our keyserver with other keyservers available today and found that our keyserver is more efficient in many ways. With the increasing use of the PGP keys in the Internet community, we expect that our keyserver will provide answers for all the issues involved.

References

[AL97] J. Allen and P. Leach. CIP transport protocols, 1997.

[Bau98] Michael Baumer. Distributed server for PGP keys synchronized by multicast. Semesterarbeit, ETH Zürich, September 1998. http://www.tik.ee.ethz.ch/tik/education/sadas/SASS1998-33/thesis.ps.gz.

[Ber] Berkeley DB. http://www.sleepycat.com/.

[CDFT98] Jon Callas, Lutz Donnerhacke, Hal Finney, and Rodney Thaler. OpenPGP message format. Internet RFC 2440, 1998.

[Eri94] H. Eriksson. MBone: The multicast backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[Fei00] Patrick Feisthammel. ThePGP Web of Trust. http://www.rubin.ch/pgp/weboftrust, 2000.

[Hor97] Marc Horowitz. PGP public key server. http://www.mit.edu/people/marc/pks/, 1997.

[Key] PGP keyserver network. http://www.pgp.net/pgpnet/.

[Ora] Oracle 8i concepts (8.1.6). http://technet.oracle.com/.

[PGP] PGP keyserver administrator's guide. http://download.nai.com/products/media/support/pgp/keyserver.pdf.

[PSVW01] Dimitris Pendarakis, Sherlia Shi, Dinesh Verma, and Marcel Waldvogel. ALMI: An application level multicast infrastructure. In *3rd USNIX Symposium on Internet Technologies and Systems (USITS '01)*, San Francisco, CA, USA, March 2001.

[Sta94] William Stallings. Pretty Good Privacy. *ConneXions*, 8(12):2–11, December 1994.

[WHK97] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). Internet RFC 2251, December 1997.