The Taste of Pesto

Feike W. Dillema and Tage Stabell-Kulø

Technical Report 2001-41

August 2001

Abstract

The Pesto distributed storage platform is geared towards a computing model where private machines play a pivotal rôle. We argue that no centralized solutions are acceptable in its design and that it supports allocation of separate tasks to separate system components found in its target environment. Hence, Pesto separates trust from responsibility, storage from access control policy, and replication from consistency control.

Pesto is designed around a few clean-cut abstractions, that make the above separations possible and efficient to implement. This report gives an overview of the main tasks typically supported by a distributed storage system, how Pesto supports these, how Pesto separates these from each other and what benefits such separation of concerns provides.

1 Introduction

"Private information is practically the source of every large, modern fortune."
Oscar Wilde, "An ideal husband", act two.

We can say that a *personal* computer is one that is available to the user when he wants to use it, while a *private* computer is one that is never available to others. We use the term *private computing* to describe the computational model which involves small, private, hand-held computers, i.e. Personal Digital Assistants (PDAs). In general, a PDA offers its user relatively few resources; a low-bandwidth user-interface, (relatively) low computational power, and highly variable communication resources. However, the PDA excels in two aspect: *Trust* and *Availability*.

A PDA that is carried by its owner has great potential to be his private computer, an attractive sanctuary for private data and computations (such as creating digital signatures). Availability means that the PDA can be used in new settings, but availability also means physical control over the device. Physical control is a prerequisite for secure operation of almost any device. With trust in the private computer itself in place, the challenge of private computing is to use a PDA as solid ground from where to extend a user's sphere of control to distributed and less-trusted resources he cares to use in a variety of different settings.

The trend towards mobile computing adds the requirement that users should be able to roam between many different environments. Independent of what

administrative domain they roam into, they want access to their personal resources while using resources available in that domain. As a general rule, we can say that policies, mechanisms and resources are different in each environment and system, and a lot of machinery between systems is necessary in order to maintain and achieve important nonfunctional aspects like security and safety across systems. However, creating this machinery is often impossible because policies and available mechanisms in the different domains are too diverse. We believe that ubiquitous computing requires a platform for distributed systems and applications that makes no assumptions on the environment the user currently is in.

There are basically two possible ways to support private computing. One can either build all the required machinery into the applications, or one can provide the applications with an underlying infrastructure. The latter approach leaves it to the applications to *adapt to* the system state as presented to them, rather than letting applications *detect* this distributed state themselves. Whether an infrastructure is viable and efficient depends on whether it supports the least common dominator of the applications' needs. We believe that most applications operating in our target environment share the need for highly available, secure storage of data.

We argued in [17] that systems for private computing should be designed in accordance with the *open-end argument*, which states that many of the nonfunctional aspects in a distributed system can only be reasonably quantified by the user (for example, who and what to trust). Qualitative assessment of information should be done by users only, and the system should be designed as to make this feasible for them to do so.

Taken together, the challenge of private computing is to design a system that allows its users to span other systems in such way that they retain ownership and control over their own information, without requiring administrative control over these systems. To achieve this goal, Pesto separates trust relations from administrative relations:

- Pesto has facilities that enable a user to control his data and enforce his real-world ownership. The user is the only one that can specify security and safety policies *and* the only one that decides who is trusted to enforce these. Pesto allows a user to delegate authority to other nodes trusted by him, but he need not do so. If the user does not authorize some part of the system to speak on his behalf on a certain matter, Pesto will involve the user himself in the decision process to resolve questions concerning it.

- Pesto supports recovery by storing and replicating all authorized updates and recording who authorized them. Experience shows that users are often unwilling to pay any non-trivial price for prevention of a risk, for as long as they have not been negatively affected by it [14]. This is well documented in the security and safety engineering literature as a very common cause for security breeches and calamities (see e.g. [1, 16]). In addition, real-world trust relationship are volatile and may change suddenly. We assume that disasters *will* happen; nodes will fail, trust will be violated, subtle mistakes will be made and accidents will occur.

- Pesto makes it possible for user Bob to share his content with another

user Alice, regardless of whether Alice is known to Pesto. She does not need to establish an administrative relationship with his administrative domain beforehand. He can delegate authority to her in the manner he feels appropriate, e.g. by using delegation certificates based on public-key cryptography. However, public-key cryptography is not a fundamental building block in the system such that delegation of authority in Pesto does not depend on it. Furthermore, Bob not only defines his access control policies but also decides who is trusted to enforce these.

- Pesto provides the means for users to share data in different settings. It is well known that sharing of data in typical file systems is relatively rare, see e.g. [11, 10]. The more private data is, sharing between different users is less likely, and one could assume that in private computing sharing would be close to non-existing. However, Pesto assumes that *the same* user, e.g. at different locations, using different machines or in different rôles, will cause private data to be shared. Also, when data *is* explicitly shared between two users, conflicts will indeed be common. All in all, conflicts will be more common than in traditional systems.

  Sharing of data in a distributed system requires some form of consistency control to resolve conflicting concurrent updates. Consistency can be enforced at the system-level or the application-level. A system-level implementation would require the user to trust a subset of the system to run a consistency control protocol on his behalf. Pesto avoids such requirements by separating replication control from consistency control.

- Pesto separates storage and availability of *data* (encrypted content) from accessibility of *content*. Assume that Alice owns some storage space. She controls this resource and decides who may make use of it. But when she grants Bob the right to use some of this storage, the *system* should not require of him that he makes the *content* available to her as well.

  Sometimes a user will have no other option however, than to trust another user with some content in order to make use of his resources, even though there might not be a basis for such trust. For example, a user roaming into unknown territory carrying merely his PDA may find himself lacking enough trusted resources to accomplish his task and his need to make progress may be greater than his need for privacy for that task. Editing a file on an untrusted machine should not require trust in other infrastructure, like the network links between that machine and the user's machine at home. In addition, it should not put the privacy of any other content of the user at risk. Pesto allows a user to put his privacy at risk in order to make work progress, while minimizing the risk involved.

- Pesto supports both disconnected and semi-disconnected operation by using asynchronous communication and computation throughout its design. Disconnected operation remains a common mode of operation for private machines, regardless of the ever growing network coverage. Also, low bandwidth and/or relatively expensive communication (like GSM data) prevails, especially for mobile machines.

In the next section we discuss related work. Following this, we present the design of the Pesto architecture and the core mechanisms of its distributed

storage infrastructure that we have designed to test our ideas on private computing. The description of our design is then followed by a section in which we discuss its features and its applicability. This discussion is followed by a section discussing Pesto implementation and context. We draw our conclusions in the final section.

2 Design

The design of Pesto carefully separates the different mechanisms a distributed storage infrastructure must support. Because of this, a user can place responsibility of the different tasks on different machines, possibly governed by a variety of administrative domains. In this way, the user is free to set up relationships he is comfortable with and assign tasks to different parts of the infrastructure. As the primary tasks in our system are providing storage, replication and access to resources, we define the responsible parts of the infrastructure accordingly:

Trusted Storage Base (TSB**)**: The set of servers a user trusts to store replicas of data (encrypted content). These servers are the storage service providers of the user. A user may define a different TSB for different sets of data.

Trusted Replicator Base (TRB**)**: The set of servers that have been given authority over a part of the user's storage resources. The servers in a TRB are trusted to enforce a replication policy on behalf of the user. Each member of a TRB is responsible for the distribution of replicas to some subset of servers in a TSB. Together they make up a directed distribution graph with edges leaving only servers in the TRB and ending in servers in the TSB. A user may define a different TRB for different sets of data.

Trusted Access Base (TAB**)**: The set of servers that have been given authority over a part of the user's content. The members of a TAB are trusted to enforce an access control policy on behalf of the user. A user may define a different TAB for different sets of data. Typically, TAB members are also members of the TRB.

In addition to this separation in space, Pesto separates tasks in time in order to support disconnected operation:

Replication: Distribution of replicas is performed using an asynchronous request-response protocol such that request and response messages may be separated by a period of disconnected operation.

Access Control: Authorization for access to content and storage can be acquired at a different time than their actual usage, allowing for work progress when disconnected from the relevant trusted access base.

The infrastructure is not responsible for consistency control. That task is separated from replication control and is left to the applications and the user himself. In addition, the infrastructure is assumed to offer best-effort service only. Monitoring and control of the quality of the actual service is left to the user and his (management) applications.

In summary, Pesto keeps different responsibilities separate, so that the user can allocate them to different, but possibly overlapping parts of the infrastructure. In addition, Pesto allows tasks to be separated in time so that users can perform the tasks requiring connectivity either before or after a period of disconnected operation. The remainder of this section describes the mechanisms that support this separation in time and space.

2.1 Administrative Domains

Each user makes up his own separate administrative domain with his private computer. A user may own other nodes that delegate all authority over their resources to the user and his private computer. Such nodes are trusted not to delegate authority over their resources to others without the user's consent. An administrative domain can thus span from anything as little as a single node to a large number of machines in a network.

A user can acquire the right to use resources at nodes outside his administrative domain and he may delegate rights to use his resources to others. We envision that users establish service contracts with other users in a variety of ways; online storage service providers offering services for a monetary fee, cooperative users exchanging storage for storage and companies offering their employees access to its storage resources. However, how such contracts are established is of no concern to Pesto.

Such a service contract could take many different forms, from signed paper contracts to electronic contracts or verbal agreements. The crux of the contract, however, is that a secret key is exchanged; the key might of course be represented as a password. Requests to store data at the remote node that are signed (encrypted) with this secret key will be honored within the rules agreed upon in the contract. Each contract is given an unique identificator (see below for a discussion on identifiers). When a request is received, the recipient uses this identifier to locate the correct contract containing the shared key that it uses to verify that the message is authentic.

2.2 Storage and Identification

Each version of a file is identified by a *globally unique identifier* (GUID) which consists of 128 randomly generated bits. Due to the random selection from an immense name space, a new GUID can be generated locally with no risk of an identical name existing anywhere in the system. It is even computationally infeasible for a malicious user to generate such duplicates. In addition, using such a large flat name space for identification has the advantage that it obviates the need for a global name service because all names are unique. Consequently, there is no need for consistency control of this name space itself.

The different versions of a file are linked together into a file version history tree. Each version of a file keeps a reference to the previous version, its parent. Concurrent updates then create file versions (each with a unique GUID) that reference the same parent, i.e. they create a branch in the version tree. This scheme facilitates recovery from unauthorized updates, and the separation of replication from consistency control.

2.3 Communication

Pesto consists of nodes that communicate with each other using the P2P protocol. Applications and supporting services that a node may host also use P2P to communicate with the storage infrastructure, but only within their node, not directly to other nodes.

The P2P protocol specifies how a node communicates with its environment, i.e. remote nodes and local applications. The P2P protocol specifies what constitutes a valid message and what constitutes a valid response to a received message. It does not specify *how* a message should be transported to its destination(s). Message transport is the task of transport services. In order to support disconnected and semi-disconnected operation, very few assumptions are made about the underlying communication infrastructure. Transport services are only assumed to implement unreliable, untrusted, best-effort transport for P2P messages over some communication channel. Each node maintains a file with local administrative information. Included in this file is a list of URLs that expresses the transport media supported by the node together with the corresponding addressing information. The GUID of this file is taken as the GUID of the node itself, and is used as communication end-point identifier.

The P2P protocol is an asynchronous request-response protocol. Each message contains a single P2P command which is either the *fetch* or the *store* command. The arguments to these commands are expressions that evaluate to GUIDs. These expressions allow for references to file versions relative to another in the file version tree, such that e.g. a fetch from a remote node of locally yet unknown versions of a file can be performed. The response, if any, to a *fetch* request is always a *store* response and vice versa. A so-called command modifier option is defined to allow specification of what part of the information tied to an argument GUID is requested:

1. The complete (encrypted) file version and its meta information.
2. Only its meta information (policy references, encrypted member key etc.).
3. A digital hash of its (encrypted) content and a supplied nonce.
4. Its encryption key.

Messages *may*, in addition to the P2P command, include information *about* files or (versions of) the files themselves. Thus file version creation (storing information about it) and file version storage (storing its actual content) can be separated in time. Only fetch requests to encryption keys are considered to be requests for access to content, all other requests are considered to be requests for access to storage resources. The messages of a request/response exchange sequence are stored in a file (each message thus has a GUID), such that they can be uniquely identified. Storing all messages in files (possibly on tertiary storage) allows for failure analysis and recovery. To fend off denial of service attacks, each message is also uniquely identified and false messages can be detected and discarded (because they can not be decrypted) while already received messages (replay attacks) can be detected and discarded because they are named [15].

Messages are encoded as MIME objects for easy parsing [3]. Together with the observation that asynchronous communication allows for the use of email

as a transport medium, this allows for the use of an existing transport infrastructure and message encoding that is also appropriate to communicate with a user himself. This way, no special provisions need to be made to incorporate individual users as communication end-points in the system.

Messages can be *guarded* in order to synchronize message processing. Guards name file versions (which may also contain other messages) that need to exist at the recipient node before the message may be processed. Guarded messages can be used to reach quorum-consensus for file updates and thus are building blocks for atomic commit protocols, for example, a "store, fetch, store" command message exchange between a set of nodes. The first message of such an exchange encapsulates only the meta information of the updates such that its recipient nodes can decide to commit or abort the update. This initial store is guarded by the final store, i.e. it will not be processed unless the final store command has been received. If a recipient is ready to commit the store request, it sends a fetch request in return for the actual content of the updates. The node that initiated the message exchange collects these responses and makes the decision to issue the final store message that encapsulates the content of the updates. This decision is based on the number of fetch responses and thus on the number of nodes that the initiator knows to be ready to commit the updates, e.g. a majority or all of the recipients. When the final store is received, it will trigger the execution of the initial store message. Execution of the initial store command results in the *creation* of the updates, and execution of the final store results in the *storage* of their data.

This basic and simple protocol is sufficient to meet our design goals for the infrastructure and it can be used as building block for application protocols with a wide variety of requirements.

2.4 Policies

As we expect the number of different policies to be much smaller than the number of files, policies are not stored as part of the files they apply to. Instead, policies are stored in files of their own. Files reference the policies that apply to them and these references are immutable and the same for all versions of a file (the content of policies themselves can change). The owner of a file is the only one that can specify what policies apply to it. To enforce this, the integrity of these policy references is protected with cryptographic means. The policy files themselves are also protected by cryptographic means as specified by their access control policy. By default, access to a policy is restricted to the members of the trusted bases (see section 1) that need access to execute the task they are trusted with. The storage resources of a node are described by storage contracts which are stored in files themselves, such that the access control mechanisms of files can also be used to enforce access control to storage. The initial storage contract of a node, describes all the node's storage resources which it delegates to the user that owns it.

2.5 Replication Policy

A replication policy specifies a set of replicas and a set of replicators. The set of replicas forms the trusted storage base (TSB) and the set of replicators forms the trusted replicator base (TRB) for the files subject to this policy. The TRB is always

a subset of the TSB for practical technical reasons without loss of generality (see below).

- A replica entry contains the identification of a node providing storage for replicas and the quality of the storage service the node is trusted to provide.
- A replicator entry contains the identification of a node that distributes replicas to other storage providing nodes, and specifies the subset of the TSB this node should distribute content to, i.e. the part of the distribution graph it is responsible for.

It is the user's task to establish the service contract and the corresponding secret key with each remote node, and to specify the members of the trusted replication base for his files. The required delegations matching the distribution graph can then be derived from these policies and can be installed by the system without further involvement of the user. For example, a user at node$_A$ establishes a service contract with node$_B$ and with node$_C$, and specifies that node$_B$ is responsible for distributing replicas to node$_C$. Node$_B$ then needs access to storage space at node$_C$, i.e. node$_B$ needs to have a secret key shared with node$_C$ to convince node$_C$ that it is authorized to distribute updates to it. As node$_A$ shares a secret key with both nodes it can securely install a new secret key that is shared between node$_B$ and node$_C$. Basically, Pesto distributes subcontracts derived from the service contracts established by the user to the member nodes of its trusted replication bases.

The whole replication policy is encrypted by default, and information about the members of the trusted replication base and the trusted storage base is only made available on a need-to-know basis. Nodes in the TRB are informed of their membership of the TRB and of the subset of nodes they are expected to distribute content to. Members of the TSB are granted merely the ability to check whether they are (still) member of this TSB. Hence, nodes cannot derive the TSB and TRB from their view on the replication policy if not explicitly granted by the user.

Although this in itself does not prohibit discovering such information by untrusted parties (using for example traffic analysis), it may significantly increase the amount of (technical, legal) resources needed to complete such a task successfully. By limiting the number of vertices in the distribution graph appropriately, some degree of protection against traffic analysis can be obtained. The user is free to put in place more elaborate schemes, if he deems it necessary [8].

2.6 Consistency

During a network partition it is impossible to know from within the minority partition, whether it is safe to proceed with a task that alters shared data [7]. Traditional, pessimistic consistency control algorithms strive to offer *single-copy* semantics, i.e. users see only a single copy at all times [12, 4]. Such strong consistency requirements are impossible or very expensive to meet in our target environment, where full connectivity might never happen. As a consequence merging of conflicting updates may be required upon reconnection.

Whether concurrent updates represent a conflict is application dependent; e.g. an append-only unordered log will never see conflicts. It is therefore our view that the system is not in a position to decide what represents a conflict and how potential conflicts should be handled. The user, on the other hand, may be able to *evaluate* the risk of disconnected operation and understand the possible consequences of his actions. It should be *his* task to decide whether being able to make work progress now is worth the risk of conflict resolution work later. It is not the task of the storage system to make such an assessment, and it will therefore not deny a user service based on some built-in notion of how to preserve correctness for application content. The system's task is merely to distribute updates. It is left to the user and his applications to define consistency policies which determine how replicas are presented to the user. Pesto task is to provide the information necessary to determine the current state of a file, such as the number of current replicas, number of versions of a file, who made these versions, and whether the file is known to be shared, so that applications can implement their consistency control policies. As described in section 2.3, applications can use the P2P protocol with its guarded messages as building block for application consistency control protocols.

Pesto allows an application to create a file version, while delaying storing its content. This can be used to implement advisory locking schemes that work well even in a semi-partitioned network; the created file version without its content functions as the advisory lock. This feature can also be used to keep track of activities in the system, i.e. nodes may store information *about* data without needing to store the actual data itself.

2.7 Authentication, Authorization, and Access Control

In general, the policies and mechanisms for authentication, authorization, and access control in Pesto are build around the concepts of delegation and handoff of authority, use of the 'speaks for' relation between principals, and secure channels that speak for principals [13].

Pesto provides the basic security mechanisms that its users use as building block to define and enforce their own security policies. In addition, Pesto allows the user to specify who else is trusted to enforce a security policy he has defined. Pesto itself is designed around a very simple security policy: all communication and all stored content is encrypted. In other words, data not properly encrypted is not accepted by the storage system. Pesto makes the encryption keys only available to the user, and leaves it to the user to decide who else should be granted access to these keys. Pesto allows users to group encryption keys that are governed by the same policy, and allows them to specify to what other nodes (if any) these should be distributed. Pesto itself does not dictate any particular security scheme to the user, but leaves it to the user to implement his own on top of the basic policy enforced by Pesto and the basic mechanisms it provides. In the remainder of this section we will first discuss how Pesto realizes its basic security policy. Then, we will discuss which mechanisms Pesto makes available to the user. Finally, we present how users may specify and enforce their own security policies on top of Pesto, using public-key certificates, delegation of authority and so on.

As discussed in section 2.1, a Pesto node typically acts as a storage provider. To obtain access to (some of) this storage, a user will use some extra-system

channel to obtain a *service contract* with this provider. The service contract will contain an encryption key; this key is installed at the server by the owner of that server. An installed secret key carries, per se, sufficient credentials to ensure that requests arriving on that channel can be authenticated as originating from the user. In practice this means that the user is given an identity at the system, that the administrator delegates authority over storage to the user, and that he asserts that the secret, shared key speaks for the user. Pesto only considers processing of requests that arrive on secure channels known to the node, i.e. all (legitimate) communication is encrypted.

Pesto keeps the complete update history of a file. An update to a file results in a new file version, and each of these versions is encrypted with a different encryption key. Such a key encrypting a version of a file is called a *version key*, and knowing such a key gives access to a single version of the file. All version keys of a file are encrypted with a *family key*; if you know this key you can get access to any version of the file.

From a Pesto node's point of view, a request (P2P message) arriving on a secure channel is authenticated if it decrypts correctly. Furthermore, a request to update an existing file (store a new file version), is assumed to have been authorized if the request is encrypted with a fresh version key, and that key is found encrypted with the family key for that file. The family key is presented to the owner of the file when the file is created; anyone who knows this key is assumed to be either the owner or someone speaking on behalf of the owner. Taken together, authentication and authorization of requests in the storage system consist of verifying the (shared key) encryption of the requests.

The version keys and the family key are also stored in a file. In order to have access to content of a file, one needs access to the file containing the keys for the file. As also this file containing keys can be replicated, a user can implement a TSB (Trusted Storage Base, nodes trusted only to store data) that is different from the TAB (Trusted Access Base, nodes that are trusted to handle content), simply by replicating the file holding keys differently from the file that is encrypted with these keys.

Because an authorized update to a file is one which is encrypted with a version key encrypted with the family key, a user can obtain authority to update a file, without getting to know the family key. For this, the owner of the file (or somebody speaking on his behalf) generates a new version key, and gives the user this version key together with that same key encrypted with the family key. As a consequence, authorization of an update can be separated in time from the actual injection of the update into the storage system. This means that a user may obtain authorization to update a file while he has connectivity to a member of the TAB and can use this authorization later, even if disconnected from the TAB (but connected to other nodes that provide storage of data only). In particular, this offers a viable implementation to the "Offline Delegation" problem [9]. This separation is facilitated by supporting separation of the creation of a new file version from the storage of its data content.

Creation of a new file version involves storing its GUID, version key (encrypted with the family key) and the credentials that were used to authorize it. Storage of the update involves then adding the data encrypted with the version key to this meta information about the new version. The task of members of the TAB is to create updates it authorizes, while the user may later store its encrypted data. If the credentials used to authorize the creation of an update

get revoked before its data is stored, the authorization of the update can be rendered harmless by storing nothing as its data as part of the revocation process. This scheme has an additional merit in that it can be used as basis for concurrency control schemes (as described in section 2.6).

Also, note that a node storing data for the user may not be trusted with the cryptographic keys needed to discriminate between authorized updates and unauthorized junk. Removal of unauthorized messages may be needed as part of a recovery process, i.e. after a node trusted to store and distribute updates got compromised and sent junk to other members of the TSB. Nodes may use P2P to monitor each other for anomalous behavior and report misbehaving nodes to the user who may then revoke the rights delegated to the misbehaving node and execute a recovery process. More detailed discussion of monitoring and recovery processes is outside the scope of this paper.

An security infrastructure solely based on shared-key encryption is not viable in a large-scale infrastructure. However, Pesto only provides integrity and privacy to the messages it transports and the content it stores. The user is free to use any existing infrastructure to realize any means of authentication and authorization he might fancy. The remainder of this section gives our example infrastructure based on SPKI [5, 6].

A user needs to establish a service contract with every node he wishes to use as storage service provider, and exchange a shared key with the provider in order to make use of the offered storage. Also, sharing of content with other users, and between the different nodes he may own requires access to and thus distribution of shared keys. Maintaining and distributing secret keys manually, is not very practical on a large scale. Instead, a higher-level authentication and authorization service can be build using public-key cryptography that can simplify such key management greatly.

A suitable tool in this respect is SPKI. Because SPKI is very flexible indeed, sophisticated 'services' such as on-line verification, revocation, and once-only semantics can be offered; see [6] for details. A user wishing to communicate with his peers on channels established by means of public-key encryption, can hand the secret keys to an application, which would then perform authentication and authorization based on chains of SPKI certificates and perform access control to content and storage on behalf of the user. A more detailed discussion is deferred to section 3 where we evaluate Pesto in general.

2.8 Local and Global Naming

At the system level, files are identified by their GUID, and files are located using the content of their replication policy. If an up-to-date replication policy cannot be located or access to it is not granted, some form of location service may be needed. In this paper we assume users sharing content know how to locate each other and may act as location service for each other in such cases. Building (persistent) publication infrastructures like GLOBE [18] on top of Pesto with (relatively) anonymous sharing of content, typically requiring a more structural approach to location services, is outside the scope of this paper.

Long random numbers are not suitable for identification of content at the user level. A file (version) may therefore be tagged by their owner with human-readable `(key, value)` pairs that are stored with its contents. Tag keys may be for example: 'name', 'version', 'collection', 'author', 'title', 'creation date'

and 'application type'. The tag value for 'name' may then for example specify a hierarchical name. The collection tag can be used to reference a file that lists the GUIDs of a collection of files. Such a 'collection' groups together a set of files with naming dependencies and defines an internally meaningful, consistent name space for the files in it. As collections are files, they can be subjected to access control and replication policies. How a user organizes and names his collections is of no concern to the system, nor for sharing between users.

3 Discussion

Our design builds on the notion of private computing and has a user-centric view in that all authority in the system originates from individual users and not from inside the system itself. The access control and replication mechanisms do not mandate any hierarchical, fixed or static structure on administrative domains. This makes our system inherently suitable for building personal ad-hoc infrastructures for sharing and cooperation between individuals, but it is certainly not limited to such. Individual administrative domains can be used as building block to construct larger domains using delegation. This requires cooperation from the individual users; users cannot be forced by the system to delegate their authority to others. Enforcement of mandatory policies within the system would require all nodes to be under full control of a single authority. But, even then, users may evade the mandated controls using channels outside the system. Extending the reach of the mandated policy to include such channels as telephone, paper notes and floppy-disks is infeasible in all environments but the most restricted ones (intelligence agencies, the military, criminal organizations and the like).

We believe that the lack of mandatory transfer of authority is not a weakness in our design, but reflects that in most real-world environments, user cooperation is a requirement to enforce a shared policy. However, cooperation from non-malicious users may be 'bought' by making the use of shared resources conditional to such cooperation. For example, a company could define a storage policy for its file servers that allows only storage of files of employees for which it has been delegated the rights to define the TAB, TRB and/or the TSB. It could then setup its communication infrastructure such that only the servers under its control may be reached through it.

Even though not listed as a design goal, our design is well suited for the construction of publication infrastructures similar to "The Eternity Service" [2] and "The XenoService" [19], that are quite resilient to denial of service attacks. Denial of service is targeted at destroying a certain resource or at exhausting the resources of the service providers. Replication of content together with logging assists in preventing the former. Resilience to resource exhaustion attacks can be gained by protecting resource use and/or by ensuring more resources are available than an attacker is able to consume. Resource protection is supported by separating access to storage from access to content and its asynchronous protocol. Its graceful degradation during periods of semi or full disconnected operation limits the damage of a successful network denial of service attack. The ability to turn secret members of a TSB into members of the public TAB with merely the exchange of a single cryptographic key, the amount of available resources can be increased dynamically during an attack

to counter the resources consumed by the attacker.

By leaving to the user to decide what credentials are considered "good enough", Pesto does not prescribe the structure and organization of the user community. For example, a "web of trust" structure constructed with PGP [20] could be used as well as some kind of certification authority hierarchy.

The separation of access to data and content implies that safety and privacy can be addressed separately; the degree of replication can be increased to obtain better availability without having to consider the trustworthiness of new storage providers. We believe that aiming at providing good end-to-end security and safety at the same time, resulted in an elegant design were the security and safety mechanisms support each other without the introduction of a lot of complexity.

4 Conclusions

Pesto's target to support private computing is responsible for most of its distinguishing properties: a flexible distributed storage system simple enough to support resource-poor devices. Pesto allows its users to specify what part of the infrastructure to trust and assumes all else is untrusted. It supports incorporation of the real-world personal and business trust relationships into the system, while not dictating or assuming such relationships in the design itself. Safety and recovery mechanisms are designed together with security mechanisms, providing ease of management and resilience against user error and violation of trust.

A dual-level encryption scheme makes read and update access control possible without relying on public-key cryptography. Public-key cryptography is supported for delegation and authorization. It is designed to perform well in networks that are semi-partitioned by supporting off-loading work to nearby, better-connected machines. Support for acquisition of authorization before actual use of it provides solid and secure support for disconnected operation.

References

[1] Ross Anderson. Why cryptosystems fail. *Communications of the ACM*, 37(11):32–41, November 1994.

[2] Ross Anderson. The Eternity service. In *Proceedings of the 1st International Conference on the Theory and Applications of Cryptology*, 1996.

[3] Nathaniel S. Borenstein and Ned Freed. MIME (multipurpose internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. RFC 1521, The Intnernet Society, September 1993.

[4] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, 1985.

[5] Carl M. Ellison. SPKI Requirements. RFC 2692, The Internet Society, September 1999.

[6] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. Rfc, The Internet Society, September 1999.

[7] David K. Gifford. Weighted voting for replicated data. In *Proceedings of 7th ACM Symposium on Operating Systems Principles*, pages 150–62. ACM Press, 1979.

[8] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communication of the ACM*, 42(2):39–41, February 1999.

[9] Arne Helme and Tage Stabell-Kulø. Offline Delegation. In *Proceedings of the 8th USENIX Security Symposium*, pages 25–33, Washington, D.C., August 1999. The USENIX Association. ISBN 1-880446-28-6.

[10] Dorota M. Huizinga and Ken A. Heflinger. Experience with connected and disconnected operation of portable notebook computers in distributed systems. In *Proceedings of the 1st IEEE Workshop on Mobile Computing Systgems and Applications*, pages 119–23. IEEE, December 1994.

[11] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[12] Leslie Lamport. On interprocess communication, part I and II. *Distributed Computing*, 1(1), 1985.

[13] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distribued systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[14] Nancy G. Leveson. *Safeware; System Safety and Computers*. Addison Wesley, 1995.

[15] Roger M. Needham. Denial of service: An example. *Communication of the ACM*, 37(11):42–46, November 1994.

[16] Peter G. Neumann. *Computer Related Risks*. Addison Wesley, Reading, Mass., 1995.

[17] Tage Stabell-Kulø, Feico Dillema, and Terje Fallmyr. The open-end argument for private computing. In Hans-W. Gellersen, editor, *Proceedings of the ACM First Symposium on Handheld, Ubiquitous Computing*, number 1707 in Lecture Notes in Computer Science, pages 124–136. Springer Verlag, October 1999.

[18] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A Wide-Area Distributed System. In *IEEE Concurrency*, pages 70–78, January 1996.

[19] Jeff Yan, Stephen Early, and Ross Anderson. The XenoService - A Distributed Defeat for Distributed Denial of Service. In *ISW 2000, IEEE computer society, Boston, USA*, October 2000.

[20] Phillip Zimmermann. *The official PGP user's guide*. The MIT Press, 1995. ISBN 0-262-74017-6.