

The Pesto Broker*

Simone Lupetti, Feike W. Dillema, Tage Stabell-Kulø
Department of Computer Science, University of Tromsø, Norway
pesto@pasta.cs.uit.no

Abstract

Pesto is a secure distributed storage systems supporting asynchronous collaboration and aiming for high availability. We present its complete architecture, focusing on the middleware layer between the storage system and applications. This layer lets applications interact with Pesto using the standard C I/O interface. This way it is possible for a wide range of applications to exploit Pesto services without any modification. Additional Pesto features (like versioning, replication and access control) that cannot be controlled through this interface are managed using two user-space utilities.

1 Introduction

We use the term private computing [1] to mean essentially the combination of mobile computing and private information. In mobile computing, the major concerns are disconnections, failures and the lack of a well defined administrative structure. Privacy on the other hand, has been mostly investigated in traditional computer system with well defined boundaries, parties and information flows. When these two worlds are combined, availability and access control must be addressed in a novel way.

The Pesto File System was conceived in order to experiment with distributed storage in the setting of private and mobile computing. We believe that the mechanisms put to use in Pesto are useful to strengthen the reliability and security of traditional computer systems.

The trust model in Pesto closely reflects the physical structure of the scenario we considered: Pesto relies on a decentralized model of trust in which resources are directly under the user's control.

The focus is on efficient use of the trusted resources available to users in order to reduce the security and safety risks of using untrusted resources in their current environment.

The decentralized structure of Pesto must cope with disconnections and node failures. Even when a resource is not available the user is always left with the possibility to make some progress.

The Pesto File System is intended as bottom layer of a full-fledged distributed file-system. To that end we have designed the Pesto Broker, a middleware layer and two management applications to complete the Pesto architecture. Combined, the middleware

*This work has been generously supported by the Research Council of Norway by means of the Arctic Bean project (IKT 2010, project number 146986/431) and the Penne project (IKTSoS project number 158569/431)

and applications are able to offer its services in a simple and standard way to existent applications. This layer works as an “adapter” from the Pesto API to commonly used interfaces for I/O.

This paper is organized as follows. In the next section the context of our work is highlighted by reminding the reader of related works in the field of distributed file system. In Section 3 the basic concepts and architecture of the Pesto File System are briefly described. Section 4 introduces the Pesto Broker describing its functional components and the services that are offered to user’s applications. Conclusion are in Section 5.

2 Related works

Andrew File System

The Andrew File System [2] is a distributed file system aiming for high scalability (up to tens of thousands of nodes). Its structure is composed of two types of hosts: *Vice* and *Virtue*. In the first category we found sever machines running trusted software, being physically secure, only managed under a central authority, etc.. They form what is commonly called the *trusted base* for a system. *Virtue* is the (typically much larger) set of client machines that, in the case of the AFS, are not trusted. No assumptions are made about their integrity.

The system heavily relies (both for scalability and performance at the client side) on client-side caching. Client-side caches only add to the performance and server scalability and they are not used to improve other properties of stored data like availability and integrity. Concurrent writes are assumed to be rare and their semantics is left undefined.

Authentication is based on the user’s password while access control is made by access control lists specifying various users (or groups of users) and, for each of them, the class of operations they may perform. Access lists are then associated with directories.

The biggest differences between Pesto and the AFS are that Pesto, being thought for personal use and relatively small forms of collaborations does not aim to scale to a large number of nodes. For the same reason the firm distinction of trusted served and untrusted clients shades in Pesto towards a flexible trust model able to adapt to the user requests (by the user of different trusted bases and fine access control).

AFS appears to the user as a directory in his local Unix file system and standard commands can be used to create subdirectories and to move, copy and delete files. Non-functional aspects like replication are hidden to the user.

Coda

Coda [3] is a file system with optimistic replication that uses AFS as underlying technology. It allows disconnected operations under the assumption that conflicting updates are a rare event. To this end Coda’s primary concern is to implement efficient and transparent *hoarding* i.e. to fill the client machine with the files expected to be accessed in the future. There is therefore a strong distinction between clients and server is maintained in the system.

In Pesto we tried instead to integrate disconnected operation in a mobile environment where some (or most) of the nodes have little amount of storage resources. Hoarding technique are in fact not suitable for nodes with these kind of features.

Also Pesto, as opposite to Coda (and its predecessor AFS), allows direct resource sharing between users without requiring any intermediary, improving this way collaboration even in presence of lack of communication or server availability in general.

Coda appears to the user as a directory in his local Unix file system. Management of non-functional aspects is done by the system administrator and is limited to define the set of servers to be used for replication.

Ficus

The Ficus [4] distributed file system targets mobile clients, and uses replication to improve availability, while weakening consistency and introducing specialized conflict resolution schemes. Replication uses *single-copy availability* so that when great availability is provided. When a replica is updated, a best effort is made to notify all the replicas that a new version of the file exist. Replicas that receive the notification then pull the new version of the file. As opposed to Pesto, Ficus add no security features beyond those of traditional Unix like file-systems, and its use is therefore restricted to sharing of files within a single administrative domain.

Plutus

Plutus [5] is a scalable cryptographic storage system that enables secure file sharing without placing much trust on the file servers. It is limited to secure storage on a single untrusted file server and does not include replication for availability. Unlike Pesto, Plutus is not a versioning file-system where data once stored is immutable. For this reason instead having to deal just with storage space taken by unauthorized writes, it has to deal with malicious users corruption and deleting files form the server. Revocation in Plutus is potentially expansive since it requires a mix or encryption, hashing and signing operations of the file (and/or its meta-data) for which access must be revoked. For mitigating this Plutus uses a lazy revocation technique where re-encryption of file data is postponed from revocation time to file update time. Meta-data still have to be revoked immediately to prevent future access by the user.

Bayou

Bayou [6] is probably the system that resembles most to Pesto. It is designed to be a platform for asynchronous collaboration and it offers weak replication for availability and support for devices with limited resources and mobile applications. Bayou's architecture is based on *servers* that store data and *clients* that are able to access data on any server to which they can communicate. A machine can be client and server at the same time.

It uses weakly consistent replication for maximum availability and a peer-to-peer algorithm called *anti-entropy* [7] in order to achieve eventual consistency. Bayou gives support for the user's application to detect and solve read-write and write-write conflicts using a semantical approach. It lets the application to decide how to deal with the inconsistency itself buy the use of merging procedures shipped along every transactions.

Bayou offers the applications a complete interface also to its non-functional aspects like replication. This implies that only applications that are aware of this can exploit Bayou's features[8].

The most striking difference with respect to Pesto is that Bayou does not try to address security, focusing instead on the efficient use of resource by the study of replication mechanisms. Like Pesto instead it tries to expose the consistency problem to applications to let them use the right ad-hoc approach to deal with it. Bayou uses merging procedures for (semi) automatic resolution while Pesto uses the Pesto Broker, described in this paper.

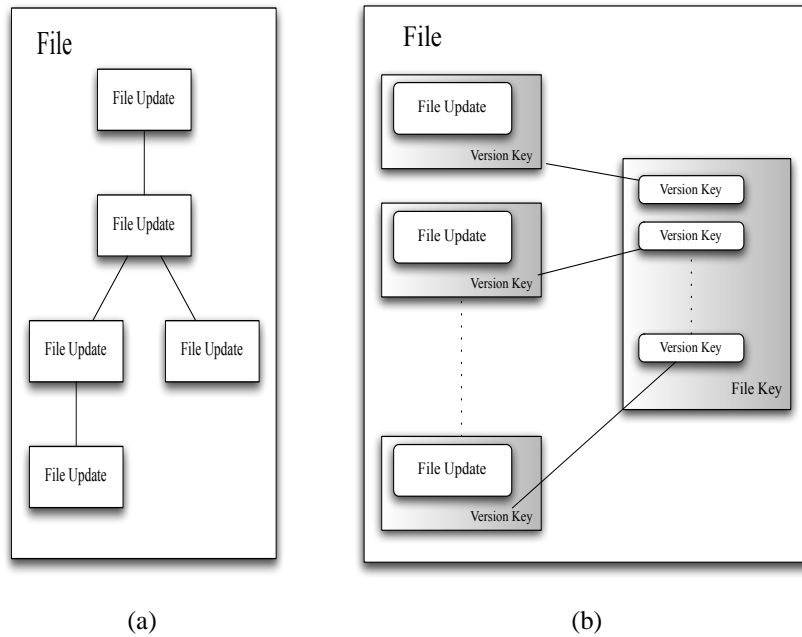


Figure 1: a) An example of a file showing the full update history of user data. Note how, after a concurrent update, the history forks in two different branches. b) Encryption scheme for a file.

3 The Pesto Architecture

The Pesto File System (PFS) is a distributed, decentralized file system. Like many distributed file systems, PFS aims at high, even ubiquitous, availability of the data it stores. In addition, it is designed to provide proper access control mechanisms, such that users can control to whom and under which circumstances their data is accessible.

The problem of loss of (good) connectivity has been addressed with the introduction of support for *disconnected operations*.

We will present only the relevant aspects of the PFS for its utilization and for the understanding of the Pesto Broker. For a closer look at the PFS architecture please refer to [9][10].

Administration framework

In private computing it is hard to imagine an efficient way to enforce centralized administration. It is instead probably more useful to envision a private computer system as a dynamic collection of distinct administrative domains than a homogeneous and well defined entity.

For this reason, in Pesto, the *user* is the unit of authority being able to define his own policies without having to rely on third parties. Each file or policy is owned by the user that created it who has full authority on them. No central authority is required, each user interacts directly with each other user in a peer-to-peer fashion.

An user's administrative domain is initially just the local node but it can be extended to (part of) other nodes by means of "agreements" with other users willing to share (some of) their resources. Pesto does not impose any predefined way to carry out these agreements.

No administrative hierarchies are present in Pesto with the only exception of a node administrator for adding and removing other users from the local node. Unlike in many other (Unix-like) operating systems, a node administrator in Pesto has no special access

privileges to the data stored by other users in the server he manages. This means that the administrator can't interfere in any way with a user's data (except by removing the user itself); encryption is used to enforce this property.

Files and Files Updates

The combination of replication and independent updates requires that the issue of consistency must be considered. In general, it is impossible to prevent inconsistencies without having to determine the global state of the system. In case of disconnection, this implies that blocking is required until the connection is re-established. To overcome this, each update is given an unique name so that one or more concurrent updates are always identifiable. Possible inconsistencies arising from concurrent updates can then be solved at application level (if required).

The system does not try to prevent an user from making updates during disconnection but rather offers support to deal with inconsistencies if and when these arise.

For these reasons user data is stored and distributed in Pesto as *file updates*. File updates are grouped in *files* (Fig. 1(a)) and adhere the WORM semantics (Write-Once, Read-Many): once stored, a file update does not change and is normally never deleted from the system. No mechanism is provided for the user to delete his or other user's files. As a consequence Pesto do not give an attacker the means to remotely harm the integrity of the system (other than the ones already present in the host system, if any).

Every modification on user data creates an update. A Pesto file is made by an initial empty version of it, together with an ordered set of all its updates. Every update consists of user data and meta-data about user data like an identifier, a link to a previous update (its parent update), encryption keys, etc. .

Concurrent updates, whether temporally or logically concurrent, refer to the same parent update.

There are no constraints on the content and the use of a file update: it is left to applications to define their own update (and access) semantics on top of the basic access to file updates provided by Pesto. For example, it is possible for an application to use differential updates while for another one it could be necessary to store the whole data contained in every update.

A file is linked to replication and access control policies. Until a file is not yet explicitly linked to a policy, it uses a safe default policy both for replication and authentication. This states that the file is stored only in the local node and that the user that created the file (and only him) is granted all types of access to it.

Names

Names in PFS are pure names [11] called *global unique identifiers* (GUID)

The name space is completely flat and it is the same one for file updates, messages, policies, and every other Pesto object. Currently a name is implemented as a string of 128 random bits. The only operation that makes sense on names is comparison for equality.

Names are solely generated by a Pesto node (in an autonomous way), never directly by users.

Maliciously generated names (with the same GUID of another object) are detected and the object with the forged name will be prevented to be used in any node where another object is present with the same name.

Pesto does not allows two object that have the same ID to be stored at the same node and thus keeps them (both as legitimate objects) well separated. We also argue that the

situation in which a randomly generated 128 bit long number is guessed by an attacker and exploited to prevent a legitimate user to use it is negligible.

Authorization

A user needs to be known by a Pesto node in order to be allowed to store (create and update) and read files at that particular site. Authentication is obtained with shared key: interaction with a (remote) node is possible only after a cryptographic key has been exchanged with it.

Pesto offers confidentiality to user data. The content of each file update is encrypted with a fresh encryption key and only those that know this *version key* have read access. The version keys of all the object updates of the same object are in turn encrypted with a *file key*. The file key is unique to each file (Fig. 1(b)).

Write access is granted to who owns a fresh version key provided that a copy of this key is found encrypted with the file key. Access control is so enforced by limiting the access to cryptographic keys: read access by limiting access to version keys and write access by limiting the access to file keys.

Authorization is enforced at file update level and read and write access are completely separated. In general there is no need to be able to read previous versions of a file to make an update to that file, for example. A higher level of access control in Pesto is provided by access control lists (ACLs) [12] that specify what user has what access rights.

Access control policies specify who, and under which conditions, is allowed to access the plaintext content of a file or, in other words, to whom version keys and/or file keys should be provided.

Delegation of access rights can be carried out outside the system. Since access is granted only based on the possession of a key, it can be delegated by simply handing-out the right key to the delegate. Note that this does not require the system to be available (nor at hand): this exchange can be made by any means available to users.

Trusted Bases

Currently, two types of policies are present in the PFS. The first specifies availability, while the second specifies the access rules for the data stored under the PFS. Availability of data is determined by the number of data replicas that are kept by the PFS on distributed servers: a replication policy specifies the replicas for a file and the method(s) of replication. The access control policy specifies accessibility (or lack thereof) of data to third parties.

What these policies really define is two sets of *trusted nodes*. These nodes are trusted only regarding specific tasks. In particular, there is a *Trusted Storage Base* (TSB) and a *Trusted Access Base* (TAB) associated with each file.

The TSB is defined as the set of nodes responsible to keep the a file replica stored and available by storing an encrypted copy of it.

The TAB is the set of nodes that are responsible for enforcing the access control to a file on behalf of its owner. The members of this set hand out the encryption keys that allow access to the content of a file (version). Different TABs can be defined for read and update access.

Every file is linked to one instance of these policies: theoretically it would be possible to have a different policy and thus trusted base for each file. The trust placed on remote nodes might vary over time and with the task at hand. A remote node can be trusted for example to hold a replica of a given file (or file update) but not for all them.

Other different trusted bases (and relative policies) can be defined by the user according his needs. For example if consistency of replicas is required it is possible to define a set of nodes responsible to vote and decide on what is the correct state of a file.

4 The Upper layers

Unlike most of other distributed storage systems (see related works), Pesto is not designed to provide a storage service as whole but to offer instead the necessary building blocks for a fully customized one. To that end our storage system offers some basic functionalities and keep them independent and well separated from each other: replication, access control, logging. Higher level aspects like consistency, authentication and recovery are completely left to the user (or the application programmer).

The Pesto Broker represents a first attempt to expose both the user and the application programmer with the Pesto services.

In order to allow preexistent software commonly in use to easily be used in conjunction with Pesto and benefit from its services we designed a light middleware layer deployed in between the PFS and the application willing to use its services.

We have chosen the standard C I/O interface as interface for the design of this layer because it is a highly standardized API and its widely adoption. As consequence the Pesto storage services are completely left transparent to the application programmer and the whole responsibility of management is putted on the user. This is not the only possible choice, but it is the only one that guarantee backward compatibility for applications.

The Pesto API is richer and more flexible than the standard C I/O library. In order to leave existing application as much as possible unmodified, some administrative task will be delegated to two *management applications*.

By using these tools the user can define his policies, import and export cryptographic keys and manage the interaction with other nodes/users. Applications instead will interface the to Pesto Broker transparently using the PFS like they would in a traditional file system.

Summing up, the Pesto Broker offers tree different interfaces to different kind of applications: one to user applications, one to a *Policy Manager* for the local Pesto node and a third one to one or more *Key Managers* dealing with the import and export of cryptographic keys (Fig. 2). The goal for the application interface is conformity with the C I/O interface while for the administrative interfaces (policy and key managers) the intention is instead simplicity and usability.

The Policy Manager is used to choose the set of storage nodes for each file (or directory). The Key Managers are instead tools able to deal with the management of cryptographic keys. Version and file keys exchange is left out from PFS so that any suitable method for this can be used. If a standardized API for importing, exporting and deploying shared keys in the PFS is provided then different methods can be used. These methods will be made available by the adoption of a different plug-ins (key managers) for each of them.

Pesto Broker

The middleware layer providing PFS services to applications is called the *Pesto Broker*. We assume here, without going into details, that Pesto offers the following interface to the Broker:

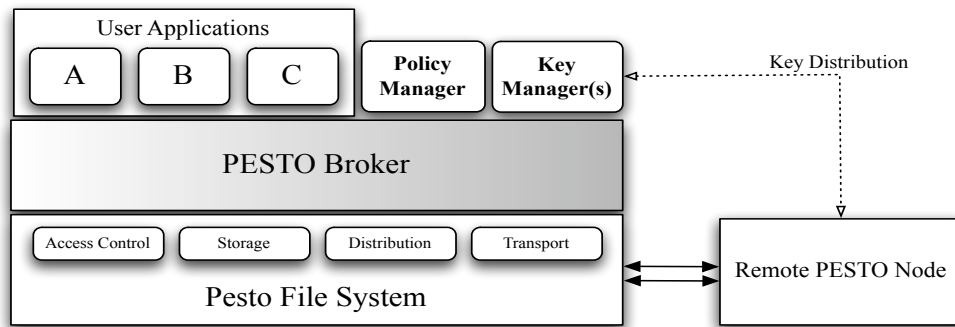


Figure 2: Multi-layered PESTO architecture.

read_file: a read operation returns a decrypted version of a stored file update

write_file: if a previous version of the file is present then an update is made. If not an entirely new file containing the user data is created and stored

write_policy: this function lets the user to define a replication and/or access policies

link_policy: a policy must be linked to one or more file to become operative. This function create this link

import_key: a cryptographic key is installed into the storage system and associated to the file or file version it encrypts for later use

read_key: a cryptographic key is read from the file system and returned

The Pesto Broker uses the user defined policies to “translate” to the standard C I/O primitives into these functions provided by the PFS.

In this process there are some issues to be considered. The first is naming. PFS only uses GUIDs for every file (or better, file version). For most of users (or applications) it is infeasible to handle files using GUIDs. This means that for many application a human-readable name must be associated with a file by linking it to the set of GUIDs of all its versions.

The Pesto Broker is responsible for translating GUIDs into file names that are meaningful to applications and vice-versa. The relation between GUIDs and file names is kept in a *directory* that, if needed, can be stored in Pesto as well (under a well known identifier for example) so that it can be replicated for availability.

Second, the semantic of a request is much more complex than the one of a standard C function call. For example versioning is an application specific requirement: the Pesto Broker must provide a way to merge it into the C I/O interface primitives. Since these cannot be modified, they will adhere the standard semantics while behind the scenes the Pesto Broker will also take in account the user defined policies about version management.

For example, when a file is closed the “close” event must be intercepted in order to trigger the creation of a new version. This only if some modifications have been made to the file against the previous version. Another example is when a read policy in Pesto states to read always a specific version of a file (this can be either the last or another one to which the user/application decide to stick).

When two logically equivalent versions of a file are present at two different sites (maybe as consequence of a disconnected modification) then, after the situation has been detected, a standard behavior can be specified for example by stating that the local version has precedence over the remote one or that the more recent one is the one to return to the user.

In order to minimize the impact on existing application, the Pesto Broker simply overrides the low level Input/Output standard C primitives using a wrapper around the C `open()` and `close()` I/O functions. In this way it is possible to leave unmodified all the other primitives like `write`, `read`, `select` and so on. An `open()` on a file will create a dummy file outside the Pesto context (a memory mapped file) and it will return its descriptor. If the `open()` has created a new Pesto file that is all. If instead an existing file is opened its name is translated into an existing update GUID and a `read_file()` is operated. On success a Pesto file version is returned and its contents (the user data) are copied to the dummy file. From then all the `write()` and `read()` (as well as all other operations) will be diverted on this dummy file. Only when a `close()` is invoked on the dummy file the content of this file is transferred to Pesto (by a `write_file()`) and stored and secured according user policies. If replication is required the PFS will take care of it informing the nodes supposed to keep a replica that a new version is available for storing.

To store the dummy file in Pesto only at closing time represent just a basic implementation of the Pesto Broker that will be enriched to allow other storage policies (like to store the dummy file after a predetermined event like the unmount of a Pesto partition from the file system).

Policy manager

File policies can be defined, modified and revoked using a management application called the Policy Manager. This uses the management API offered by the Pesto Broker and provides an intuitive way for the user to interact with the Pesto policies. In particular it allows other users to define replication and access policies (their trusted bases) and to bind these policies to files (or set of files).

At the moment, the possibility to define new trusted bases for user-defined tasks is not exploited in the Pesto Broker. This task requires careful planning and probably a direct interaction with the Pesto File System as a whole.

Replication policies are defined in terms of which nodes are required to store (keep a replica of) a file. Note that according the Pesto semantics a replication set may be defined by more than one node: the local node defines just an initial set of replicators. The replication policy by default is itself replicated to the replicator set it defines. A replication policy has a tree structure in which, starting from the root (the local node), one branch for every replicator forks. The leaves of the tree constitute the set of replicators. Every member of the storage base is also allowed to modify the branch of the replication policy containing itself adding its own nodes to the policy and extending the availability of the data behind the initial user expectations.

Access control policies can be defined along the same lines as replication policies.

Since a policy is itself a file, it is stored according the WORM semantic. This means that also policies are versioned: unless a policy is not already existent, every `write_policy` function call creates an update of that policy. A policy can not be revoked, it can only be updated. The only way to annul the effect of a replication/access policy is to create a new empty update to it. As consequence, every file linked to that policy from now on will use a default policy: access without restrictions is granted only to the creator of the file and

the replicator set includes only the local node.

Summing up the Policy Manager gives the possibility to define a *preset configuration* for:

- The trusted storage bases
- The trusted access bases
- Other settings like write/update policy, versioning policy, reference nodes, etc.

A list of patches is kept by the policy manager. Every file or directory can be linked to a (and only one) specific patch and will therefore use all its settings. The link between the file and “its” patch is partially kept in the PFS itself (link to storage and access control policies) and partially as application data in the Policy Manager.

Key Manager(s)

The local node must share a cryptographic key at least with each node in its storage bases for authentication purposes. How these keys are exchanged has been intentionally left out of scope of the PFS. This for essentially two reasons: The first is that any suitable method for key exchange could possibly be used for it.

Sending a shared key by secure e-mail [13], using an implementations of Diffie-Hellman key exchange protocol [14], by SMS messages or by traditional mail should be equally feasible in order to obtain as much flexibility as possible.

Second, many of these tools can be used at the same time and replaced at any moment giving high reliability and resistance to failures. More than one different type of key manager can run in parallel on a single Pesto node.

The Pesto Broker has a well defined interface to import and export tools to connect to Pesto and fetch/deploy cryptographic keys for authentication by the use of the `import_key` and `read_key` PFS primitives. These tools will be available as stand-alone programs and existing tools can be reused as well.

We believe that in a heterogeneous environment containing personal devices, leaving the user or the system administrator to choose the tool that fits better the real needs for key exchange is desirable. On the other hand since simplicity of use is one of the main goal of Pesto, some mechanism for easily importing and exporting other’s node key need be present. From this considerations the choice for a well defined API.

5 Conclusions and future work

Pesto is a distributed storage system designed for asynchronous collaboration. It offers some basic functionalities like replication, access control and versioning to be used to build a complete, personalized storage system. The basic functions and mechanisms are controlled and managed by the user who specifies his policies to the underlying system. This requires some additional support both for the end user and the application programmer and, in general, can be done in two way: the first one is to empower the application programmer to manage all the non-functional aspect of Pesto while the second one is to let the the user to completely manage them.

We have chosen the second approach in order to allow existing applications to use the Pesto functionality without any needs for modification.

To this end the Pesto Broker splits functional and non-functional aspects in two different set of API: the first one caters for the application’s programmer and focus on

backward compatibility for functional aspects. The second instead caters for the user and focus on usability and non-functional aspects. On top of the Pesto Broker we find two kinds of management applications: a Policy Manager for the user to define, modify and revoke his policies, and one or more Key Managers letting him to import and export keys from other nodes using a variety of methods. These applications are designed to help the user to deal with non-functional aspects of Pesto and are completely independent both from Pesto and the Pesto Broker. This means that they can be easily replaced, modified or built in as part of other software without affecting the Pesto Broker nor Pesto.

The resulting architecture resembles the application server/API combination common in many middleware platforms ([15] for example) where instead of components we have stand-alone applications.

We envision for example new Pesto-aware applications able to exploit both sets of API (for functional and non-functional aspects) and providing even more control to the user.

The Pesto File System has been completely implemented on a FreeBSD operative system while the Broker is still under development. We have also planned to embed the Pesto as database service in an application server and to rewrite its API to be accessible to the deployed components. The Pesto Broker (with the management applications) represents therefore both a first step towards a more sound solution for the management of the Pesto services and a complement to the PFS in order to make it immediately useful for applications.

With the Pesto broker we also intend to investigate the trade-off between compatibility, transparency for the application programmer and transparency for the user in the use of the Pesto File System. We believe that this trade-off needs to be investigated further in order to provide the user with an interface that is at the same time the most intuitive and the most powerful.

References

- [1] Stabell-Kulø, T., Dillema, F., Fallmyr, T.: The open-end argument for private computing. In Gellersen, H.W., ed.: Proceedings of the ACM First Symposium on Handheld, Ubiquitous Computing. Number 1707 in Lecture Notes in Computer Science, Springer Verlag (1999) 124–136
- [2] Howard, J.: An overview of the Andrew File System. In USENIX Association, ed.: USENIX Conference Proceedings (Dallas, TX, USA). (1988) 23–26
- [3] Kistler, J., Satyanarayanan, M.: Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems* **10** (1992) 3–25
- [4] Guy, R.: Ficus: A Very Large Scale Reliable Distributed File System. Ph.D. dissertation, University of California, Los Angeles (1991)
- [5] Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., Fu, K.: Plutus — scalable secure file sharing on untrusted storage. In USENIX Association, ed.: Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST). (2003)
- [6] Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Welch, B.B.: The bayou architecture: Support for data sharing among mobile users. In: Proceedings

IEEE Workshop on Mobile Computing Systems & Applications, Santa Cruz, California (1994) 2–7

- [7] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, New York, NY, USA, ACM Press (1987) 1–12
- [8] Terry, D.B., Petersen, K., Spreitzer, M.J., Theimer, M.M.: The case for non-transparent replication: Examples from bayou. In: IEEE Data Engineering, IEEE Computer Society (1998) 12–20
- [9] Dillema, F., Stabell-Kulø, T.: Pesto flavoured security. In: 22nd Symp. on Reliable Distributed Systems (SRDS), Florence, Italy, IEEE Computer Society Press, Los Alamitos, California (2003) 241–249
- [10] Dillema, F.W.: Disconnected Operation in the Pesto Storage System. PhD thesis, University of Tromsø (2005)
- [11] Needham, R.M.: Names. In Mullender, S.J., ed.: Distributed Systems. ACM Press (1989) 89–102
- [12] Lampson, B.: Protection. In: Proceedings of the 5th Annual Princeton Conference on Information Sciences and Systems, Princeton University (1971) 437–443
- [13] Zimmermann, P.: The official PGP user's guide. The MIT Press, Cambridge, Massachusetts, 02142 USA (1995)
- [14] Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory **IT-22** (1976) 644–654
- [15] DeMichiel, L.G.: Enterprise Java Beans specification, version 2.1 (2003)