

The Open-End Argument and the New File Repository

Feico Dillema and Tage Stabell-Kulø

Department of Computer Science, University of Tromsø, Norway
{feico,tage}@pasta.cs.uit.no

Abstract. A truly personal machine, called a private machine and implemented as a Personal Digital Assistant (PDA), is fundamentally different from traditional machines. It is *personal* and *private* in an unprecedented manner, and its modus operandi is such that network and power failures will not be rare. Designing distributed systems where PDAs are treated as “first class citizens” is a challenge. Furthermore, private assets (electronic money, keys for authentication and opening doors) will be stored in PDAs. Ownership and control of these assets and the media that store and communicate them should remain with the user. This must be reflected in the design of systems for private computing. We introduce the “open-ended argument” to describe the design strategy we used for designing a system that is designed to reveal information to the user (as opposed to hide it). We argue and show that when systems are designed this way, the user (a human) is better able to control the system and his personal data, as he can make better decisions than the system itself based on *qualitative assessment* of the provided information. The system, called New File Repository or NFR, we have designed and implemented under this design guidelines is presented and discussed.

1 Introduction

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

We have designed and implemented a system around the use of Personal Digital Assistants (PDAs). While designing the system, we came to realize the need for attention to where decisions are taken as the system is operating. Users’ ability to exercise control depends on whether the system is structured so that no important decision is taken without first consulting them. Moreover, important information for making decisions should be made available in spite of the engineering tradition of information hiding. A traditionally layered system shields the user from the intrinsic details of the system, and does not require him to be competent to act according to the information the system might present about its internal state. Removing such requirements from the user is generally considered as an advantage, in particular in order to make the system user-friendly.

Systems based on this model are designed to be transparent. The “transparency design principle” states that users should be shielded as much as possible from the inner workings and state of the system they are using (see, for example [3]). This principle is not only applied to end-systems, but also more generally to layers of abstraction inside the system. While this design principle is important for building abstractions and has been applied successfully to many system designs, its rigid application suffers from some serious problems.

The end-to-end argument [8] argues against placement of functional components and services at low levels of abstraction in a computer system. A function or service should only be implemented at a (low-level) architectural layer if it is needed by all clients of that layer and if it can be completely implemented by that layer. In general, a layer cannot handle all types of errors, and will in some situations enter undesired states, such as for instance by blocking or even failure by crashing. Such errors or exceptional events in one layer require intervention from the layer above, where the peer entities may utilize their position to resolve the situation below. This may propagate up through the layers until it reaches the topmost layer, which implements the interface towards the user. In the end, the user is confronted with a malfunctioning system.

With the advent of truly personal and mobile computing and communication devices, we believe the system and user model as described earlier is rendered ineffective, for at least three reasons:

- We believe that “real world” private assets will be stored in PDAs, owned and controlled by individual users. Such assets can be diaries, keys for authentication and/or access control, and possibly electronic money. Users will want zealous control over such resources; before valuables are handed to the system, a user needs to be able to trust it. Trust and trust relations are crucial components in any secure system. Trust is a *feeling*, based on personal experience, social context and personal perception. It can not be measured or quantified by the system in any way, and it is often difficult or impossible for a user to quantify (and specify to the system) *a priori*. Hence, it can not be used in algorithmic computations by the system for making decisions and evaluations. In essence, acknowledging the non-computability of trust implies that systems designed with the traditional system and user model, cannot support the user’s perception of trust and cannot make decisions based on a user’s trust and trust relations.
- The user will view the system as providing him with a set of services for processing, storage, retrieval and communication of (private) data. Traditionally, the system decides for the user how such services treat the user’s data. In addition, the system is often the only designated authority when it comes to making qualitative judgments (based on policies specified by the system maintainer/owner). This makes sense in traditional environments where no part of the system is really owned by the actual user and the user’s data is not considered (at least by the owner of the system) as his private property. When systems are to process a user’s truly private data, the user will want to be in control of the system rather than being “just a user”.

- PDAs often operate at the limits of what can be achieved by current technology and operational problems due to resource shortage must be considered part of normal operation. Since PDAs typically are powered by batteries, communicate over wireless links and may roam in hostile environments, several types of failure are likely to occur much more frequently than in non-PDA based systems. Almost all failures will occur in what is normally considered as “infrastructure”, and users are not supposed to interact with problems at such a low level. However, when operating a PDA each and every user must be involved also in these issues in order to keep the system usable even when parts of the system fail as part of almost normal daily operation.

There is a common denominator in these three aspects. Distributed systems are designed as layers of abstraction, with peer entities communicating with each other over the network. The user, however, often has no peer in computer systems (in the “other end” so to speak) that can replace the user for making certain decisions. We have an *open-ended* setting when the users rôle is controlling rather than just using the system.

Taken together, systems that encompass PDAs do not fit elegantly into the classical model of distributed systems design, where computation may be distributed, but where authority and competence to make decisions is centralized. Based on our experiences, we propose a new design principle, named the “open-end argument”. In the following section, Section 2, we argue for and describe this “open-end argument” for systems design. The design and implementation of a system under guidance of this argument — The New File Repository — is described in Section 3. Conclusions and future work is offered in Section 6.

2 The open-end argument

Rigid application of the transparency design principle tends to structure the design in ways that causes two problems:

- A quest for transparency tends to push complexity down into the lower layers because more functionality is needed to handle ever more exceptions. As a consequence, hiding the state of the system from users may in itself result in increased system complexity. Increased complexity often leads to more complex failure modes of the system at large, which in effect may be increasingly difficult to hide from users, and so on. For example, NFS (Network File System [9]) tries to shield users from the fact that some files are remote. It provides location transparency, and users are presented with a uniform interface for local and remote files. However, an application that issues a write operation on what seems to be a local file can get in return an error message like “Remote Host Unavailable”, or simply block (depending on implementation details). Maintaining transparency over failures like this would require the addition in the system of significantly more (and more complex) machinery, like (write) caching and consistency control and so on.

- Application of the transparency principle seems to structure the system such that the control users have over the system is reduced. This is especially true

when it is combined with a rather pessimistic user model in which users are not assumed to be competent to make any decisions based on the state of the system. In such systems, problems that the system cannot handle transparently and which the typical user of the system is not assumed to be able to handle either, are left to be resolved by the more knowledgeable system manager or help-desk support personnel. The position of the user in the system can then be said to be on the “edges” of the system. When the user is placed at the edge of the system, avoiding the potential confusion caused by strange error messages in NFS (example above), is no requirement for the system design. However, proper handling of failures is of greater concern in PDA-based systems as argued in the introduction of this paper.

End-to-end and similar style arguments have led to the idea of “stupid operating systems”, “stupid networks” and “stupid processors” [7]. We introduce the idea of “stupid PDAs” in order to discuss the structure of systems that support owners of such machines. A stupid PDA is designed to be simple enough for its owner to feel confident he understands how the PDA operates (on his behalf!) and for him to feel able to control the PDA. It keeps its owner well-informed about relevant aspects of its (internal) state at all times, rather than hiding it from him. The essence of what is known about the state of the system should be communicated. Only then is it possible for the user to understand what is going on, and intervene if necessary. Moreover, to aid understanding and control, it is a design challenge to reduce the number of visible and thus important states in the system, while at the same time convey as much information as is required.

Although PDAs can be viewed as just another access point to distributed systems, treating them as such in systems design ignores many important properties. The most important property of a PDA is that its user (and owner) exercises physical control over it. This might seem obvious, but it is the main discriminating factor between a PDA and a desktop PC or workstation. This property, when properly exploited by the systems’ design, yields a computing and communication device that none but its user controls. This facilitates *private computing*.¹ Private computing implies that the owner of a private computer is in full control of it. For the owner to be confident that he fully controls the activities of and access to this private computer, systems need to be explicitly designed for it. “User friendliness” achieved only by applying the transparency design principle is insufficient or even inappropriate for such systems. Hiding the true activities and state of the system from the owner inevitably result in reduced confidence on his ability to control his private machine.

Confidence is a human feeling and is therefore difficult or impossible for others to quantify. Like concepts such as “trust” and “competence” it is defined by a user’s beliefs, experiences and personality (amongst others). These notions are inherently qualitative in nature and must be quantified in a meaningful way in order to be used in a computation. The notion of trust can serve as an example. Alice might trust Bob in some circumstances, but not in others. Furthermore,

¹ In order to avoid confusion with “personal computing” denoting one (desktop) machine per user (PC), we will use the term *private computing* instead.

whether she trusts Bob depends on the credentials he conveys with his requests to Alice. She may accept an authentication performed by a Kerberos server for access to her work-related documents [10], but require a completely different set of credentials for access to private documents [6]. Alice makes a qualitative assessment of the information presented to her. For example: Is this signature ‘better’ than the other, and do I find it ‘sufficient’ for this particular request?. Such assessments are well known in “the real world”, where different credentials (i.e. papers) are needed for different purposes; a passport is needed in some situations, an ID card suffices in others. Different papers that “say” essentially the same thing, but with different quality. Again, it is the notion of private computing using PDAs that introduces such qualitative assessments into the system. This forces the designer to either ignore the user, or ensure that the user can control the system.

Often a system is considered easy-to-use if it requires little a priori knowledge and little training from the user. Even though competent users can not be assumed for many system designs, we may assume that even ignorant users have the ability to learn. Unfortunately, many system designs hide much of their inner workings (by design), so that it is very difficult for a user to learn how to control and manage it better. We argue that especially in the context of private computing, a system that is not designed to let a users learn to control and manage it, can not be considered easy-to-use as users are prevented from learning how to deal with situations the system itself can not resolve transparently. In order to support user learning and user control, a private computing system should provide the user with all the system information he needs and/or desires.

We believe that a different approach is required when designing systems to support private computing. In addition we argue that only the user can and should be the arbiter of what is important for him to control. This can not be left for the system to decide. We call this line of argument *the Open-end argument*:

Open-end argument: The system should be designed in such a way that in all situations where qualitative assessment of information is needed to make a decision, the user is informed and consulted.

This argument guides a system designer in deciding when to follow the transparency design principle and when to violate it by informing and consulting the end-user. The remaining of this article describes a system designed under guidance of the open-end argument.

3 The NFR Storage Architecture

The primary task of NFR is to provide a distributed storage infrastructure to its users. Like many distributed (file) systems, NFR 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 its data is accessible.

Design decisions for NFR are made with the Open-End Argument as design guideline, which is the primary reason for most of NFRs discriminating features. In practice it means that NFR ‘merely’ tries to enforce user-specified policies, instead of incorporating and enforcing policies of its own. In many cases, user policies can be specified *a priori*, but it is often infeasible for a user to specify policies that cover all possible states the system can be in. Typically, it is appropriate and feasible for a user to specify its policies in advance for the common case, i.e. when everything works as expected (e.g. no partial failure due to temporary resource shortage). For exceptional states of the system, either intended or unintended, it is often infeasible for the user to pre-specify a policy that covers handling of these states, especially when the number of these states and their causes are large. Most systems therefore do not ask users to pre-specify policies for such cases.

The common approach to handle exceptional states in a system, is to let the system follow pre-defined built-in internal system policy (e.g. built-in error-handling) and/or simply report failure and deny its users service until the system enters a normal state of operation again (like e.g. service is denied while some host or network link is down). In contrast, NFR is designed to be policy free; it will not fall back in internal built-in policies to handle states that the user has not or could not anticipate when specifying its policies to NFR. NFR is designed to *consult the user* when policy decisions need to be made that are not covered by the policy the user specified a priori. Instead of denying the user service, NFR leaves it to the discretion and wisdom of the user to decide if and how to proceed.

Currently, two types of policies are enforced by NFR. The first specifies availability, while the second specifies accessibility of data stored by NFR. Availability of data is determined by the number of replicas of the data that are kept by NFR on distributed servers, and the replication policy specifies the (number of) replicas and the method(s) of replication. Access control policies in NFR specify accessibility (or lack thereof) of data to users and third parties. NFR is designed around five core architectural abstractions to provide this functionality; Files, Modules, Safes, Locks and Users. Each File, Module, Safe, Lock and User is identified by its globally unique Name in NFR. The following sections describe each of these abstractions and their semantics in NFR. Note that we write File, Name etc. with an initial capital letter to refer to the NFR abstractions, and use file and name and the like in lowercase to refer to their respective general counterpart.

3.1 Names

Names in NFR are pure names, i.e. they are purely used for identification purposes and contain no structure. Name spaces in NFR are therefore completely flat, and no additional contextual information is required to interpret the name (i.e. bind the name to the object it refers to). Names in NFR are therefore globally unique. Currently, a Name is implemented as a random bitstring 128 bits long. Names are generated solely by NFR servers, never by users of NFR. To users,

a Name is merely a handle to reference a particular object in NFR. Users and their applications may implement name spaces with stronger semantics on top of NFRs flat name space. The NULL Name is defined in NFR as a reference to the non-existent or empty object. As Names are picked at random from a very large number space (2 to the power of 128), the chance to pick a new Name that is identical to an existing one is negligible. This is of course also true for the NULL Name consisting of 128 zero bits, such that it is safe to tie a special meaning to this Name.

3.2 Files

The unit of storage in NFR is a File. A File is like the traditional file abstraction in that it represents a sequence of bytes and no type is associated with its contents. It differs from the traditional in the operations that are defined on it and their semantics. A File has read, write and delete operations defined on it, but as a File is the *unit* of storage there is no counterpart in NFR of the traditional open and close file operations. Also, as File Names are generated by NFR there are no file name manipulation operations defined on Files (like rename, move and link).

The semantics of read, write and delete operations on Files differ significantly from their traditional counterparts. Most notably, Files have write-once, read-many times semantics. Existing Files in NFR can be ‘written’ to, but the semantics of this are that a new File is created with a new Name (that is returned to the User as the result of the write operation), instead of the existing File and its contents being overwritten. In NFR, the newly created File is called a *child* of the existing File that was written to. A File can be ‘written to’ multiple times and can therefor have more than a single child, and a child File can of course have children of its own. Each File is then part of a family tree of Files. The only semantics of this family tree in NFR are that a child File was created by a write operation on its parent and the root File of a tree was created by a writing to the (non-existent) File with the NULL Name. Only whole family trees can be deleted at once in NFR, individual Files can not be removed from their respective tree; a delete operation on a File deletes all other Files in its family tree also.

The presence of multiple children of a File will be notified by NFR to the user (asynchronously), so that the user (or his application) may decide whether this represents a conflict in his view and if so, resolve it as he sees fit. This is in accordance with the Open-End Argument as NFR tries to execute the wishes of the user as best as possible, but will notify the user when it does not have complete information available to do so.

3.3 Modules

A Module is the unit of replication in NFR; it is a replication policy specification constructed by a User. Every File in NFR is associated with exactly one Module, although this may be the ‘empty’ Module which specifies the no-replication

policy (store in local NFR server only). The empty Module is referenced with the NULL Name, and does not need to be created explicitly by a User. All Files in a particular File family tree are associated with ('in') the same Module; changing the Module association of a File ('moving a File to another Module') therefore also changes it for all its ancestors and siblings.

Currently, a replication policy in NFR specifies the NFR servers that should store replicas of a Module (i.e. the Files in it). In addition, it specifies how these NFR servers can be reached, in particular what transport protocol to use for communicating with each NFR server. It is important to note that a Module only specifies a replication policy, and does *not* specify a policy for concurrency control. The separation of replication and concurrency control policy is enabled by the write-once, read-many times semantics of Files. Concurrent writes in NFR produce multiple independent child Files instead of potentially inconsistent copies of the same File. The semantics of multiple children of a File are irrelevant for Modules; each is just a File that is subject to the same replication policy.

Modules have write-once, read-many times semantics just like Files, and modifying a Module produces a new child Module with a new Name. Three operations on a Module are defined that can modify it; change of its policy, moving a File (i.e. its whole family tree) into and the moving of a File out of a Module (either as result of File deletion or by moving it to another Module). In a Module family tree it is then also possible that multiple children exist for a particular Module. As Files are only allowed to be in exactly one Module at a time, this may represent an illegal state with undefined semantics. When NFR has two user-specified replication policies for a File, it needs to resolve which one is the correct one before it can progress in enforcing the replication policy for that File. In such a case, NFR will make an 'educated guess'² or make a random choice to what the user's intentions are. The User (or Users) will be notified of the conflict and how NFR chose to resolve it, so that the User may correct NFR's choice in case it guessed wrong. Rationale behind this is that on the short term progress of policy enforcement is more important than enforcing the correct policy at all times when it comes to replication.

3.4 Safes

A Safe is the unit of access control in NFR. Like Modules, Safes are a policy specification constructed by a User; a policy for access control for the Files and/or Modules 'in' the Safe. Each File and Module in NFR is associated with at least one Safe, although this may be the 'empty' Safe which specifies the no-access policy (only the owner/creator of the File/Module has access rights, nobody else). The empty Safe is referenced with the NULL Name, and does not need to be created by a User.

² For example by looking at the local timestamp on each policy to establish an ordering between them. As NFR does not assume the local clocks to be globally synchronized this is an educated guess at best.

All Files and Modules part of a particular family tree are always associated with ('in') the same Safe. In other words, a single Safe specifies the access control policy for a whole family tree. Access can be granted for individual Files (Name-based access control) or by their relative position in their family tree. For example, the policy of a Safe can specify that access to leaf children is granted to a particular User, but no access is given to parents in a tree. This way, a User does not need to re-specify the access control policy when he adds a new child to a tree (by writing to the parent); the child in a sense inherits its policy from its parent. Also, it enables a User to e.g. specify more limited access to old versions of a File than to the most recent³ Note however that at any point in time different NFR servers may have a different view on what a leaf child is in the tree, as the latest children may not have been received by all replicas yet. The User should be aware of this when he specifies his access control policy.

Safes have write-once, read-many times semantics just like Files and Modules; modifying a Safe produces a new child Safe with a new Name. Three operations on a Safe are defined that can modify it; change of its policy, moving a File/Module (i.e. its whole family tree) into and the moving of a File/Module out of a Safe (either as result of File deletion or by moving it to another Safe). In a Safe family tree it is then also possible that multiple children exist for a particular Safe. When two children of a Safe have equal access control policies, NFR will simply merge the two Safes into a single new one. When the access control policies differ this may mean a File is in multiple Safes. Unlike with Modules, this is not an illegal state in NFR. The semantics of a File (or Module) being in multiple Safes is the union of their access control lists. However, in the case of multiple children of a single parent Safe it may very well not be exactly what the User intended. His intention may have been to create a single Safe with the union of the access control lists of both children instead. Therefore, NFR will notify the User (asynchronously) when it discovers multiple children on a Safe such that the User can decide whether this was intended or not (and initiate an explicit merge on the two Safes to unify them to clean things up).

In addition to Files and Modules, a Safe can also contain other Safes or even itself (the default). In other words, the Safe itself can also be subjected to an access control policy. Currently, the Safe specifies access rights for read-access, write-access (i.e. creating a new child in the tree), delete-access and for (un-)locking and delegation. Locks are described in the next section and their use is discussed in section 4.

3.5 Locks

A Lock is the basic building block for concurrency control in NFR. Locks do *not* provide any concurrency control by themselves; they are merely a tool NFR provides for implementing concurrency control policies by Users and their applications. Locks therefore, are called such only because they are provided by NFR

³ The other way around is next expected to be particularly meaningful, but is currently not considered illegal in NFR

to be used as such, not because they actually provide any locking. A Lock is merely a piece of information tied to e.g. a File, it doesn't prevent or block any type of operation on that File.

The difference between 'ordinary' Files and Locks is in how they are replicated by NFR. The replication policy of a Module in NFR is enforced on a best-effort basis; after a new File is stored in an NFR server, NFR starts to distribute it according the replication policy of the Module of that File. However, NFR gives no service guarantees to how many replicas actually exist at any point in time (at least one, i.e. the local one exists). For Locks NFR *does* give guarantees to the number of replicas that exist if a Lock is created. The User specifies the minimum number of replicas for a Lock to exist. NFR will not create the Lock before it can meet this specified replication requirement.

Currently, NFR supports two types of Locks that differ only in the minimum number of replicas that are required for their existence. The first type requires the Lock to be present at a majority of all replicas as specified in the replication policy. The second type requires the Lock to be present at all replicas for it to be created at all. Locks can be associated with anything that is associated with a replication policy, i.e. anything that is in a Module like Files and Safes but also Modules themselves (which specify the replication policy for themselves, i.e. a Module is 'in' itself).

When a Lock is associated with e.g. a particular File it will also be associated with all those siblings of that File the Lock owner creates. In a sense, a Lock follows the actions (writes to a File) of the User that set the Lock. A Lock will not prevent any other User from for example writing to the same File and thus create a child of it. The only operation another User is prevented from doing is setting the same⁴Lock on that File or on any of its siblings.

In addition to setting/creating and releasing/destroying a Lock, a User can read a Lock. This means not only that a User can detect the presence of a Lock in NFR, but also that the User can obtain an external representation of it that can e.g. be communicated to other Users of NFR or even third parties. Such an external representation and identification of a Lock in NFR is like the concept of a capability [1]. By sharing such a capability with another user, the lock can be controlled and released by any one of them. This usage of capabilities to represent Locks outside NFR resembles the way Amoeba used them as a form of shared secret [11]. The important issue is that *the User* controls the lifespan and semantics of his Lock and controls who may manipulate a Lock he created, not NFR.

3.6 Users

A User is the unit of authority in NFR. Each File, Module and Safe is owned by the User that created it. A User has full authority over what it has created/stored

⁴ Note that NFR does not prevent setting two different Locks on the same File. Whether such an action makes sense and has semantics defined for it is up to the User and/or his application program.

in NFR. Access control is User based, i.e. an access control list specifies what User has what access rights. But the User that has full authority over (owns it) a File etc. can delegate (part of) his authority to other Users, but also to users/third parties not known to NFR itself. Also, an access control policy can specify such delegation rights for other Users.

A User needs to exist (be known by) an NFR server in order for it to be allowed to store its data at that particular NFR server. The User with the NULL Name is special in that it always exist and is the only one that is allowed to add/remove Users to a NFR server. This NULL User is local and specific to each NFR server and it can be regarded as the system administrator of that NFR server. Note however, that unlike in many other (Unix-like operating) systems, this system administrator has no special access privileges to the data stored in that server by its normal Users. It merely exercises control over those NFR server resources that it owns, i.e. storage space and not the actual data stored in it.

4 Concurrency control

Locks in NFR can be used by Users (and their applications) to implement a concurrency control policy. A Lock might be set on e.g. a File to ensure consistency. Locking is the ultimate pessimistic concurrency control regime. In NFR, however, a lock on a file is not absolute, in that any User might ignore any Lock at will. The semantics are that when a locked File is read from NFR by a User that does not control the Lock, a warning notification describing the Lock is issued to him. If such a User attempts to write to a locked File, the data is “redirected” onto what can be interpreted as a “shadow child” of the original File and the owner/creator of the Lock on the (parent) File is notified of the presence of such a shadow child. In this interpretation a “primary child” is a child created by a User that controls the Lock on the parent File, and which will also be locked by the Lock associated with its parent. Note that a shadow child may exist for a locked File even though no primary child exists (yet).

A User that controls the Lock may decide to upgrade a shadow child to the primary child by locking it with the same Lock as its parent. Alternatively such a User may merge a shadow child and the primary somehow into a new locked (i.e. primary) child of that primary (by a write operation on the primary child). Note that such a merging operation is outside the scope of NFR itself as NFR has no business or means to define the semantics of such an operation. It needs to be defined and executed by the User or the application he uses.

The important issue is that the user may ignore a lock because he has extra-system information about whether it is safe to do so. “Safe” in this context means that creating a shadow child of that locked File, will not produce two inconsistent versions (children) of that File such that the shadow child can simply be denoted as primary later by a User that controls the Lock. A conversation or phone call with a co-worker (“Are you really editing this file at the moment?”) are typical examples of channels for information outside the system. But even if the user has no such information he may decide to take his chances and proceed

“unsafely” and create a shadow child. He may then risk having to clean up later (merging children). It is *his* task to decide though whether being able to make work progress now is worth this risk of extra work later. It is not the task of NFR to make such an assessment, and NFR therefor will not deny a user service based on some built-in notion of how to preserve correctness.

During a network partition it is impossible to know from within the system and/or application in the minority partition, whether it is safe to proceed with a task that alters shared data [2]. In fact, it is even impossible to know whether a datum has become shared at all. The user, on the other hand, may be capable of understanding the situation and *evaluate* the risk, and even more important, understand the consequences of his actions. Also, as mentioned above, the user might use channels outside the system to gather information for such understanding and evaluation.

For example, consider the user Alice, who has a computer (NFR server) in her office, on which the latest up-to-date version (child) of a file is stored; Alice is at home. During the evening she decides to edit the file, but her requests for the latest version can not be fulfilled due to a network partition, i.e. she is unable to contact the NFR server that stores it. Alice is now left to *decide* which actions the system shall undertake on her behalf. It is obvious that she can safely proceed working at home if she knows the state of the file, and can refrain from altering those sections where she knows her out-of-date version differs from the more up-to-date version at work. By exploiting her understanding of the situation she is able to take the “correct” action even though seen from the traditional systems’ point of view, she has created two inconsistent versions (children) of the file. When the network becomes operational again, the version in Alice’s office is augmented with the “shadow” she created at home. The shadow contains her updates, and she can either merge the two into a new version, replace one of them (i.e. denote the one or the other as primary). The point is, NFR did not force her to try to circumvent the system services, but rather made it possible for her to obtain progress even when she was in a minority partition. She obtained progress at the risk of conflicting updates, but at her discretion.

NFR can also be employed by different users wanting to share data. Consider the following scenario: Alice and Bob share a file they cooperatively edit while they are employed at different sites. On one fine Sunday afternoon Alice decides to write, she contacts her local NFR server and requests a lock on the latest version of the file to inform Bob that he should refrain from editing that file while she is working on it. The NFR response can be that a copy of the file is available (after all, it is replicated) but the network is partitioned between the sites of Alice and Bob, so that NFR is not able to inform Bob about Alice’s intentions of editing the file, nor can NFR guarantee Alice that the version of the file she holds is actually the latest (Bob may already have created a new version of which no replica exists yet at Alice’s site). Alice is left to make the decision how to proceed based on her extra-system knowledge, e.g. about the work habits of Bob and the amount of work involved in merging possible conflicts later. It is obvious that she has no guarantee from the system whatsoever, but

she decides whether to proceed or not. She might use for example the telephone to establish a de facto lock by means of social engineering, to reduce the risk of inconsistencies. Also in this scenario, NFR supports smooth integration when connectivity is restored. NFR will inform all relevant users about the current state of the file when possible such that they can take proper if it is not in the intended state.

5 Access Control

Users are known by their public keys, and a User is represented in the system as a delegation from a key to some form of authenticated channel; access control is based on access control lists (ACLs) [5,6] as described in section 3.4. The system's rôle is to implement the policy set by the User. In particular, NFR has no business in authenticating Users, but should only validate requests for access.

NFR protects the interests of users by implementing whatever policy the User specifies. A User can, at any time, create a certificate that will give another user access to one of his files stored by NFR. That other user need not be known to NFR in advance, i.e. be represented as a User in NFR. When *the user* decides what credential are sufficient for access to his files, useful services such as offline delegation become feasible [4]. Offline delegation enables a user to grant somebody access to his own files without having to interact with NFR itself. Delegation certificates generated offline have the same "value" as those generated in cooperation with an NFR server, since the user is the focal point in the system as dictated by the Open-End Argument.

6 Conclusions

Personal Digital Assistants are becoming commonplace, and they will become more resourceful and can be used in an increasing number of ways. A machine that is trusted can be part of daily life, and their owners will soon come to depend on them for their real-life daily activities. Successfully including such machines in a static infrastructure requires that the design acknowledges the special properties PDAs have, and the special way PDAs can and will be used. We have defined and discussed the open-end argument. It states that whenever assessing quality of information is required for making a decision, the user should be consulted. We claim that systems structured in accordance with the open-end argument will be different than systems targeted at support for static systems only. Decentralization of control, management and authority inevitably leads to new semantics of terms such as "conflicting updates" and "trusted".

We presented the New File Repository. It serves as a research vehicle for investigation of the open-end argument and its effect and usefulness in systems design. By firmly placing the user in the control of the system, the New File Repository has been designed to gracefully support users with PDAs. In our

setting with concurrency control and access control we reveal that an open-ended system like NFR shows different characteristics than systems designed using more traditional system design guidelines.

The security infrastructure built to support users using NFR is currently being used to monitor access to other objects than files. In particular, we are designing and building a system for access control (to a physical location) based on the assumption that users have private machines at hand. Also this design effort is guided, naturally, by the open-end argument.

Acknowledgments

Terje Fallmyr, Frode Fjeld, Åge Kvalnes have given us valuable feedback that has improved the presentation of our ideas in this paper. The design of NFR, New File Repository, is based to a large extent on on experience gained with its predecessor called the File Repository. Our gratitude therefor also goes out to all who were involved in creating it and gave us feedback on it. Arne Helme participated in the work on offline delegation. Finally, thanks to all current and former residents of the “PASTA laboratory” for creating a stimulating environment for us.

References

1. J. B. Dennis and E. C. van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, March 1966.
2. D. K. Gifford. Weighted voting for replicated data. In *Proceedings of 7th SOSP*, pages 150–62. ACM Press, 1979.
3. A. Goscinski. *Distributed Operating Systems, The Logical Design*. Addison-Wesley, 1991.
4. A. Helme and T. Stabell-Kulø. Offline delegation. In *8th Usenix Security Symposium*, 1999. Accepted for publication.
5. B. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in *Operating Systems Review*, 8, 1, January 1974, pp. 18–24.
6. B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.
7. D. P. Reed, J. H. Saltzer, and D. D. Clark. Active networking and end-to-end arguments. *IEEE Network*, 12(3):69–71, May 1998.
8. J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
9. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Summer conference proceedings, Portland 1985: June 11–14, 1985, Portland, Oregon USA*, pages 119–130. USENIX, Summer 1985.
10. J. G. Steiner, B. G. Neumann, and J. I. Schiller. Kerberos: An Authentication System for Open Network Systems. In *Proc. of the Winter 1988 Usenix Conference*, pages 191–201, February 1988.

11. A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. van Rossum. Experiences with the Amoeba distributed operating system. *Communication of the ACM*, 33(12):46–63, December 1990.