# Black Box Leases

John R. Douceur and Jon Howell

MSR TR-2005-120
TR release date: September 20, 2005
Patent filing date: December 17, 2004

## 1 Abstract

Conventional leases protect consistent access to a single unit of state shared among a set of computers. For some types of data, however, this lease exposes too much information, reducing security, and limiting performance by limiting concurrency. We describe a form of lease called a *black-box lease* that is specialized to the semantics of the data type to which shared access is provided. This specialization exposes between clients only that information required for the given data type, otherwise hiding information to improve concurrency and security.

## 2 Origin of this report

This technical report describes an invention upon which a patent has been filed. One of the purposes of the patent system is to encourage the public disclosure of new inventions; unfortunately, patent applications are written in "Patentese," a degenerate descendent of English notorious for its inscrutability.

Filed patent applications in Patentese are often derived from reasonably scrutible original documents written in English, and that is the case for the present invention. The authors have no immediate further plans for this invention, and hence do not expect to write a pedagogically-improved description any time soon. Therefore, we are releasing this English document as a technical report in the hopes of better communicating the invention to the technical public.

Therefore, this report has been produced with a minimum of effort beyond that originally required to begin the Patentese disclosure. In particular, it does not evaluate an implementation, and is decidedly incomplete about academic scholarship. *Caveat lector*.

Section 3 provides background on the key technologies used in the invention. Section 4 describes the invention itself.

## 3 Background

Section 3.1 describes the context of the invention, the Farsite file system. Section 3.2 describes leases in general. Section 3.3 describes the limitations of the conventional form of lease.

### *3.1 Farsite*

Farsite is a serverless distributed file system. Logically, it acts like a central file server, but physically, all computation, communication, and storage are distributed among the client computers in the system. Some or all of the client machines act together in small

groups to perform functions analogous to those performed by a server in a conventional server-based distributed file system. For purposes of the present document, such groups will be referred to as "servers." In part, this is because the present invention can be used in a conventional server-based distributed file system, as well as a fully distributed, serverless file system such as Farsite [1].

For performance reasons, it is advantageous for distributed file systems to perform a large fraction of their operations directly on the client machines that initiate the operations, rather than sending each operation individually back to a server for remote processing. However, this presents a problem in keeping the file-system data consistent: If two clients make conflicting changes to the file-system state at the same time, the system enters an inconsistent state. One approach that is sometimes used to deal with this problem is to have resolution mechanisms that attempt to rectify inconsistencies after they occur; thus, the system is allowed to enter an inconsistent state and is subsequently corrected, to whatever extent is possible. A completely different approach is to prevent the inconsistency in the first place, by restricting the operations performed by each client to those that do not conflict with operations performed by other clients. Farsite takes the latter approach.

## *3.2 Leases*

Farsite protects consistency by means of leases [2]. A lease grants a client permission to observe or modify a particular subset of the global file-system data fields. For example, a client may receive a lease that permits it to modify the contents of a particular file; this is commonly called a "write lease." Alternatively, it may receive a lease that permits it to observe the contents of a particular file; this is commonly called a "read lease." For consistency, if any client is granted a write lease on the contents of a particular file, no other client may be granted a read lease on the contents of that same file. By contrast, it is permissible for multiple clients to have read leases on the contents of a single file. This is known as "single writer, multiple reader" or SWMR semantics.

When a client attempts an operation, it does not necessarily know what leases it may need for the operation. For example, Windows file-system semantics specify that a directory may not be deleted if it contains any files or subdirectories (these are commonly called "children" of the directory). Thus, if a directory has no children (and if several other conditions are met), the correct response to a delete-directory operation is to delete the directory and return a success code. On the other hand, if a directory has at least one child, the correct response to a delete-directory operation is to return an error code and not delete the directory. The client that is attempting the delete-directory operation may not initially know whether the directory has any children, so it does not know whether it requires a read lease on every child field (all indicating the absence of a child), or if requires a read lease on one child field (indicating the presence of a child). Furthermore, since there are other conditions that must hold for this operation to succeed, it might be that no child leases are required at all, because the operation will fail for an unrelated reason. We therefore say that client lease requests often have "proof-or-counterexample" semantics. By this we mean that the server is obligated to provide either proof that an operation will succeed (by issuing all required leases necessary for the successful completion of the operation) or a counterexample showing at least one cause for the

operation to fail (this is typically a single read lease). Since the server generally has more information than the client, it is in a better position than the client to determine which leases the client needs.

## 3.3 Shared-value leases and their problems

Traditionally, if any data field in a distributed file system is leased to clients, the field is protected by a SWMR lease or a degenerate form thereof. Degenerate forms include read-only leases, which protect data that clients are never allowed to modify, and single read/write leases, which do not allow multiple clients even for read access. We refer to all these various classes of leases as "shared-value leases," because the lease governs access to a single value that is shared among all clients.

Shared values represent a poor abstraction for some file-system data fields. One example is the set of handles that a particular client has open on a file, which needs to be a protected value for observing Windows deletion semantics. In a Windows-compatible file-system, file deletion occurs with the following procedure: A client opens a handle to a file and uses this handle to set the "deletion disposition" bit on the file. Once the deletion disposition is set, no new handles may be opened on the file. Existing handles on the file can be closed normally, except that when the last handle is closed, the file is unlinked from the file-system directory structure.

To perform deletion in a distributed file system, when a client closes a handle on a file, the client must know whether it is closing the last handle that any client has open on the file. If a SWMR lease were used for the set of handles open on a file, deletion would proceed as follows: When a client closes its locally last handle on a file, and if the deletion disposition (protected by a SWMR lease) is set, then the client must have either a read lease on every other client's handle-set fields (all indicating the absence of an open handle on the file) or a read lease on one particular client's handle-set field (indicating the presence of at least one open handle on the file). With this information, the client can determine whether it should unlink the file.

Using a shared-value lease in this application causes two problems: security and performance. Security suffers because a client can see which handles other clients have open on a file, which is information they should not be privy to. Performance suffers because when some client X holds a read lease on the handles open by another client Y, client Y is unable to concurrently hold a write lease that would enable it to change the handles it has open.

A simple improvement can partially address the above problems: Rather than having the SWMR lease protect the set of handles open on a file, the SWMR lease can protect a Boolean value that indicates whether the set of handles is non-empty. This improves security because a client cannot see which specific handles another client has open on a file, and it improves performance because a client does not need a write lease to modify the set of handles it has open on a file unless the set is transiting between empty and non-empty. However, security still suffers because a client can see which other clients have any handles open on the file. Furthermore, performance still suffers because when some client X holds a read lease on the non-emptiness of client Y's handle set, client Y is

unable to close its last handle, even though some other client Z might have a handle open, which should be sufficient for client X to know that it needn't unlink the file.

A better approach would be to provide an abstraction that hides unnecessary information for security reasons and allows greater concurrency because the system can permit changes to values that do not affect the observable parts of the abstraction. The present invention provides such a better approach.

# 4 The invention: Black-box leases

The present invention is a mechanism for protecting and sharing information in a distributed system. Unlike the traditional "shared-value leases" that are used for this purpose, the present invention introduces "black-box leases," in which values that are particular to distinct clients are partitioned among clients by the lease mechanism itself, rather than in an ad hoc manner on top of a shared-value lease mechanism. A black-box lease presents to each client an aggregation of other clients' values.

## *4.1 Leases in the Farsite system*

In the Farsite system, shared-value leases are used to protect the following fields of each file:

- The file's deletion disposition
- The file's parent directory
- The file's contents (if the file is not a directory)
- Each child of the file (if the file is a directory)

The latter case requires an infinite number of shared-value leases. There is one such lease for each label that could be bound to a child file. For example, there is a lease governing the child bound to the label "A", and another lease governing the child bound to the label "B", and so forth. For those labels that are not bound to any child, the value of the associated field is nil. Since there are an infinite number of leases and fields, Farsite uses a very compact representation of this data, so it can be stored in a finite and reasonably sized space. We do not describe the details here, since this is not part of the present invention.

The Farsite system uses black-box leases to protect the following fields of each file:

- The handles that a client has open on the file
- The modes for which a client has the file open (these will be described in more detail below)

## *4.2 Black-box lease structure: SelfValue and OtherValue*

As mentioned above, a black-box lease exposes to each client an aggregation of other clients' values. For each data field protected by a black-box lease, each client has two sub-fields: SelfValue and OtherValue. The SelfValue field holds the client-specific value

for the particular client, and the OtherValue field holds the aggregation of all other clients' SelfValues, as illustrated in Fig. A. In the general case, these values can be of any type, and the aggregation function in the black box can be any function. The clients do not need to know what the function is; they only need to know what rules they should follow when using the OtherValues that they observe.

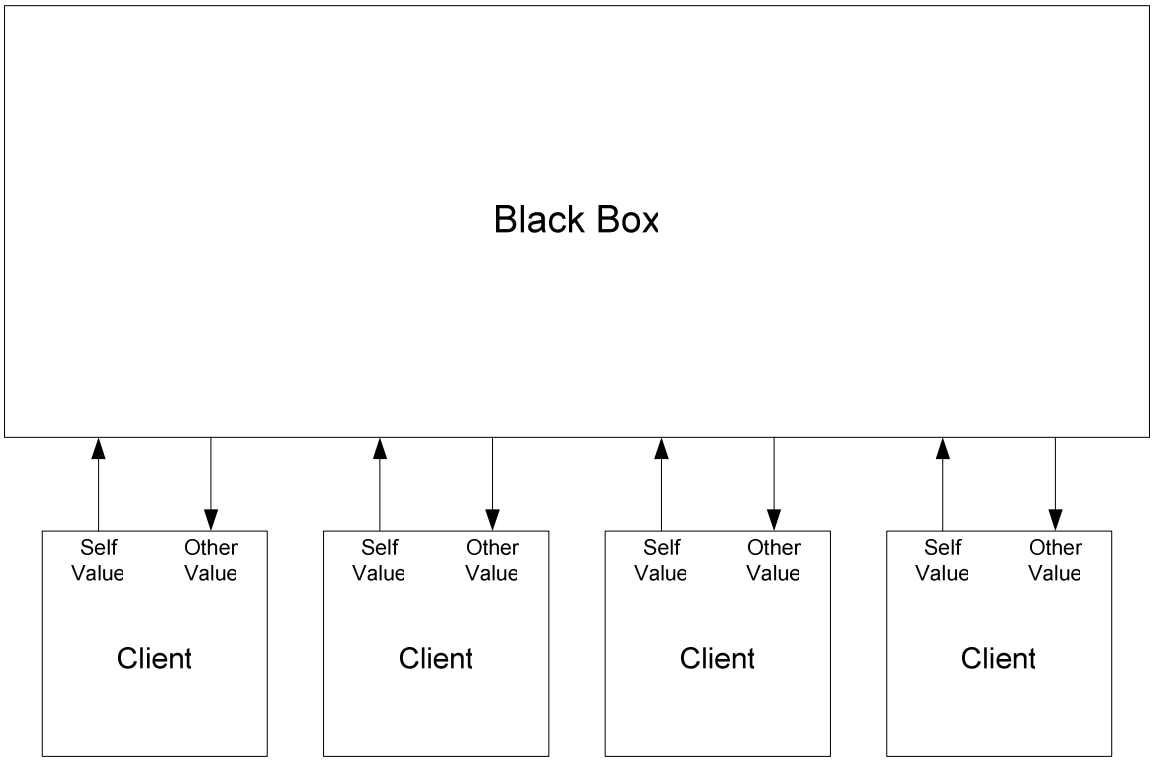


*Figure A. Black-box lease structure.*

Farsite uses black-box leases for seven distinct fields of each file; however, all seven of these fields use black-boxes with the same types and the same aggregation functions. The SelfValue and OtherValue fields for each black-box lease have Boolean type, and the aggregation function is the disjunction of all other clients' values, as shown in Fig. B.

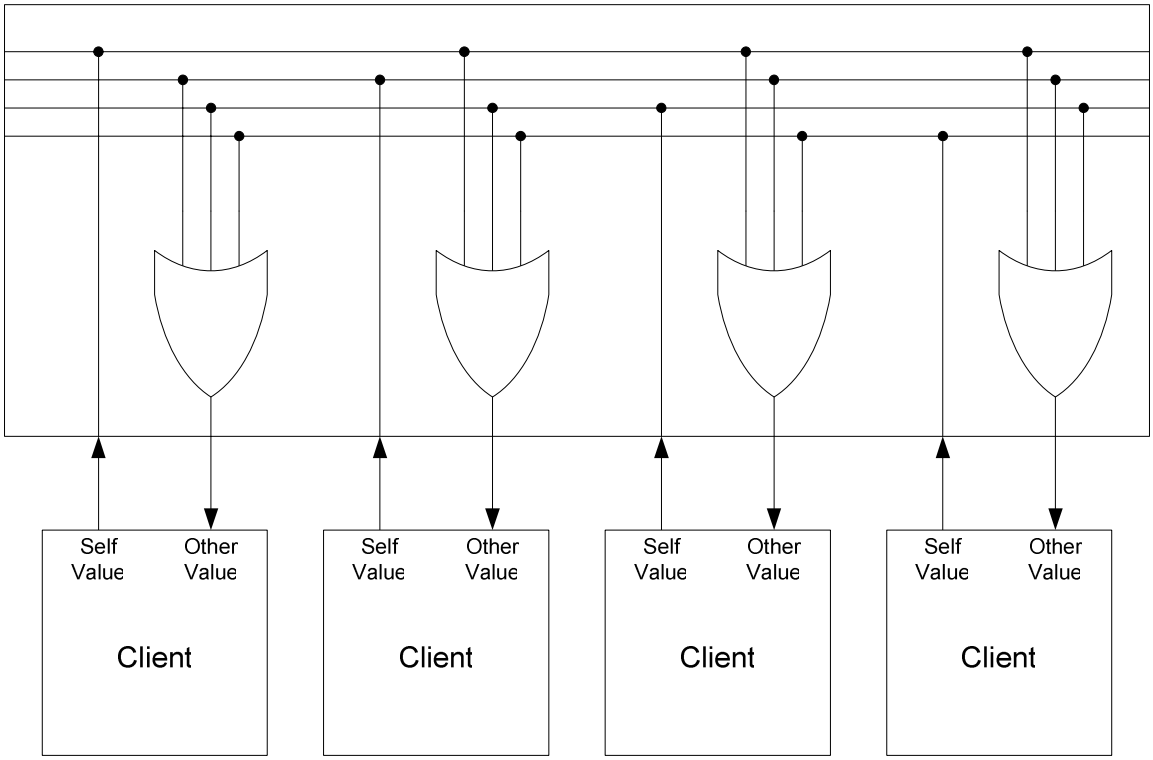


*Figure B. A black-box lease of Boolean type.*

## *4.3 Black-box leases for open handles*

By way of example, each file has a black-box lease that is associated with the set of handles each client has open on the file. When a client opens its first handle on a file, it sets its SelfValue for this black-box lease to True. If the client opens more handles on the file, the client does not change its SelfValue. When the client closes handles on the file, the client does not change its SelfValue unless it is closing the last of its handles, in which case it sets its SelfValue to False. A client can determine whether any other client has a SelfValue of True by looking at its own OtherValue. If its OtherValue is True, then at least one other client has a SelfValue of True; if its OtherValue is False, then all other clients have a SelfValue of False. This follows from the use of a disjunction as an aggregation function.

Revisiting the deletion example described above in the Introduction, the black-box lease allows the deletion to proceed as follows: When a client closes its locally last handle on a file, and if the deletion disposition (protected by a shared-value lease) is set, then the client looks at the OtherValue sub-field of the black-box lease associated with the open handles on the file. If the OtherValue is False, then no other clients have open handles on the file, and the client should unlink the file. If the OtherValue is True, then at least one other client has at least one open handle on the file, and the client should not unlink the file.

Observe how the black-box lease solves the security and performance issues that arose when using shared-value leases for clients' open handles. There is no security violation because the client cannot see which other clients have handles open on the file. There is

no performance problem because no other client is prevented from closing its last handle on the file, unless it is the very last handle held by any other client in the entire system. Furthermore, note that the black-box lease inherently provides the proof-or-counterexample semantics described in the Introduction.

## 4.4 Black-box leases for modes

When a handle is opened on a file in Windows, it is opened for a specific set of access flags and sharing flags. A file's contents can be read via a handle only if the handle has been opened with the AccessRead flag; a file's contents can be written via a handle only if the handle has been opened with the AccessWrite flag; and a file's deletion disposition can be set via a handle only if the handle has been opened with the AccessDelete flag. The sharing flags specify what access flags are permitted to other handles open on the same file. A handle cannot be opened with the AccessRead flag if another handle is open on the same file and the other handle has not been opened with the ShareRead flag. Similarly, a handle cannot be opened with the AccessWrite or AccessDelete flags if any other handle open on the file was not opened with the ShareWrite or ShareDelete flag, respectively.

In Farsite, we found it simpler to describe file-sharing behavior with flags that have the inverse meaning from those in Windows. It is a simple matter to negate the sharing flags to produce the three flags that are used internal to Farsite: ExcludeRead, ExcludeWrite, and ExcludeDelete. We refer to the three access flags and three exclude flags collectively as "mode flags."

The rules for determining mode conflicts are as follows, where X is Read, Write, or Delete:

- A handle cannot be opened for AccessX if any other handle on the file is open for ExcludeX.
- A handle cannot be opened for ExcludeX if any other handle on the file is open for AccessX.

It is a simple matter to implement these semantics using black-box leases. For each mode, each file has a black-box lease that is associated with the set of handles each client has open on the file with the given mode. A client can determine whether any other client has a conflicting mode by looking at its OtherValue for the mode. If its OtherValue is True, then at least one other client has a handle open with the given mode; if its OtherValue is False, then no other clients have handles open with the given mode.

## 4.5 Black-box lease protection: SelfWrite and OtherRead

Each client's SelfValue is protected by a SelfWrite lease. The client is allowed to change its SelfValue from True to False, or from False to True, if and only if it holds the associated SelfWrite lease. Each client's OtherValue is protected by an OtherRead lease. The client is able to observe the Boolean state of its OtherValue if and only if it holds the associated OtherRead lease.

## *4.6 Processing OtherRead requests*

A server may grant an OtherRead lease to a client under either of two conditions: (1) At least one other client has SelfValue that is stuck True, or (2) all other clients have SelfValues that are stuck False. In this context, "stuck" means that the other client does not hold a SelfWrite lease, and thus it is not permitted to change its SelfValue. If a client has SelfValue that is stuck True, then any other client can be granted an OtherRead lease with an OtherValue of True. If all clients other than some client X have SelfValues that are stuck False (irrespective of the SelfWrite and SelfValue of client X), then client X can be granted an OtherRead lease with an OtherValue of False.

Fig. C shows a state diagram that illustrates the possible behaviors of a server in response to a request by client X for an OtherRead lease. In state C-1, some client Y (which is not the same as client X) does not hold a SelfWrite lease, and its SelfValue is True; in this state, the server is enabled to grant client X an OtherRead lease with an OtherValue of True. In state C-2, all clients other than client X do not hold SelfWrite leases, and their SelfValues are all False; in this state, the server is enabled to grant client X an OtherRead lease with an OtherValue of False. In state C-3, neither of the above conditions is satisfied; there is therefore at least one client Y (not the same as client X) that holds a SelfWrite lease. In this state, the server is enabled to recall the SelfWrite lease from client Y. When client Y returns its SelfWrite lease, it informs the server of its associated SelfValue. If client Y's SelfValue is True, then the state changes to C-1, and the server can grant the OtherRead lease to client X. If client Y's SelfValue is False, then the state will change to C-2 if no other clients hold SelfWrite leases, and the state will remain in C-3 if at least one other client holds a SelfWrite lease.

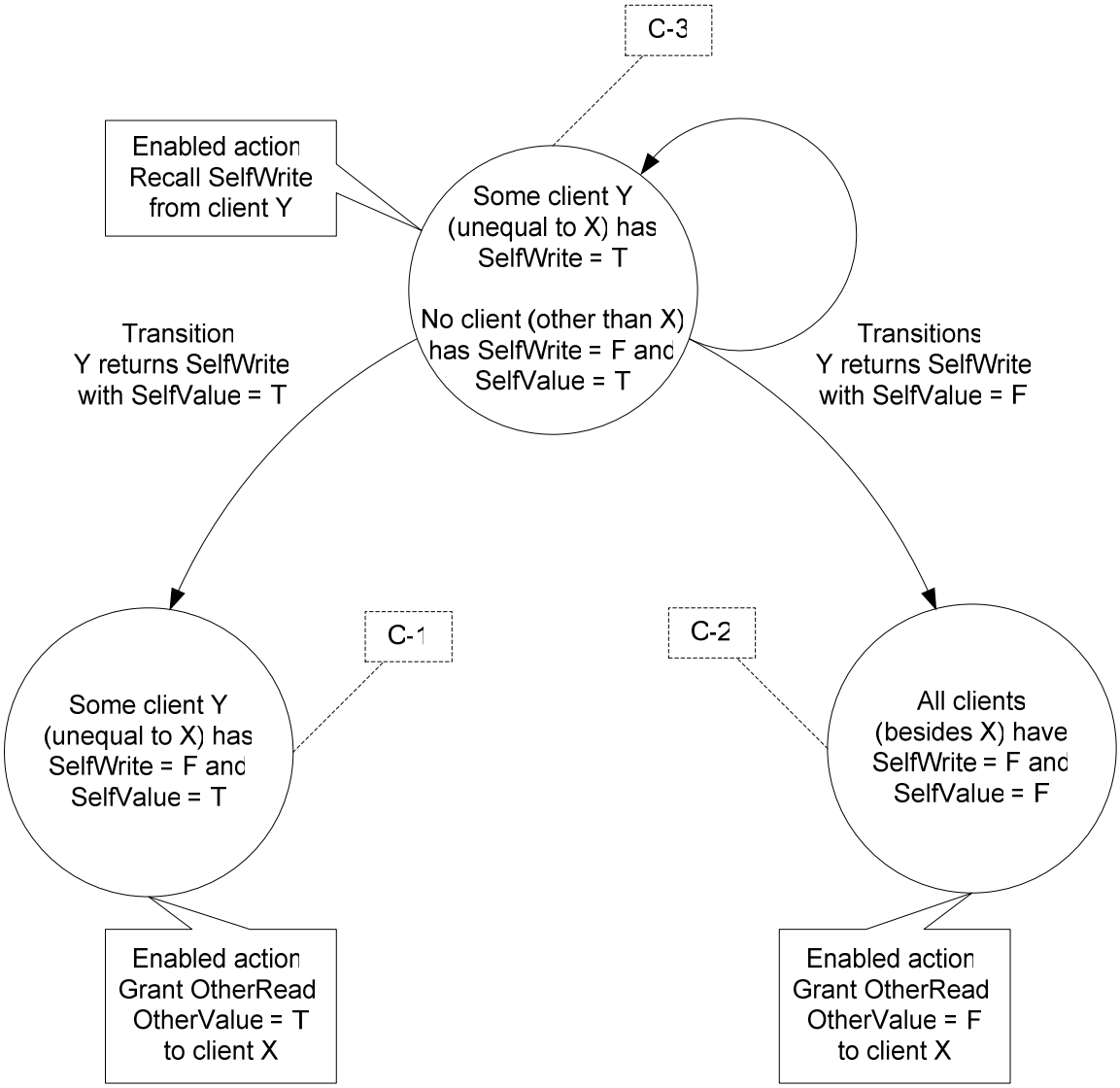


*Figure C. The behaviors of a server in response to a request for an OtherRead lease*

## *4.7 Processing SelfWrite requests*

The conditions under which a server may grant a SelfWrite lease to a client are somewhat more involved than those for granting an OtherRead lease. A server can grant a SelfWrite lease to client X if every other client either "ignores" client X's SelfValue or is "shielded" from client X's SelfValue. A client Y is said to ignore client X's SelfValue if client Y does not have an OtherRead lease; therefore, it will not be able to observe any changes to client X's SelfValue. A client Y is said to be shielded from client X's SelfValue if some third client Z has a SelfValue that is stuck True; since the aggregation function presents the disjunction of all other client's SelfValues, a stuck True value of client Z will cause client Y to see an OtherValue of True, irrespective of the value of client X's SelfValue. In such a case, we refer to client Z as a "shield" for client X's SelfValue.

Fig. D shows a state diagram that illustrates the possible behaviors of a server in response to a request by client X for a SelfWrite lease. In state D-1, the server is enabled to grant a SelfWrite lease to client X, because every client Y that does not ignore client X's SelfValue is shielded by some third client Z whose SelfValue is stuck True. Note that it is possible for two clients to act as shields for each other; however, it is not possible for a client to act as a shield for itself, since the aggregation function for each client does not include that client's own SelfValue.

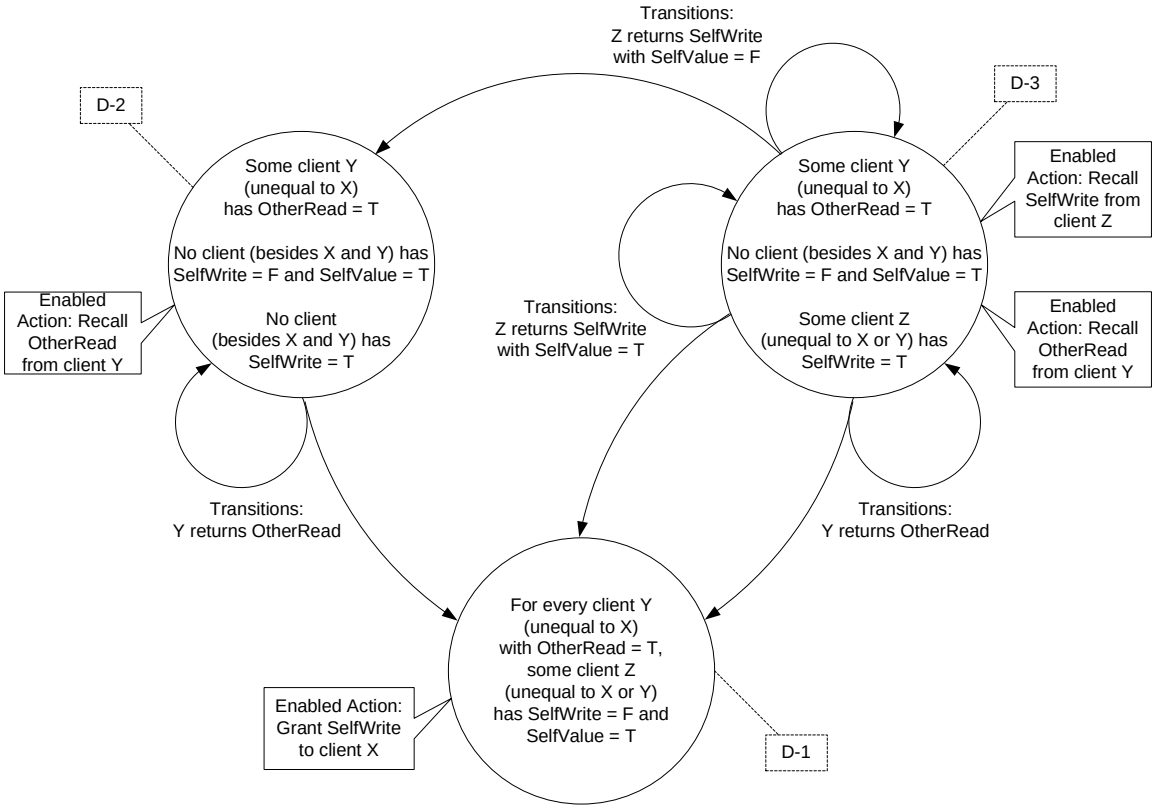


*Figure D. The behaviors of a server in response to a request for an SelfWrite lease*

Three conditions hold in state D-2: (1) At least one client Y is not ignoring client X's SelfValue; (2) no other client shields client Y from the SelfValue of client X; and (3) there are no other clients that could potentially become shields, because no other client has a SelfWrite lease that could be recalled. Therefore, the only action enabled in this state is for the server to recall the OtherRead lease from client Y. When client Y returns its OtherRead lease, the state will change to D-1 if no other clients hold OtherRead leases, and the state will remain in D-2 if at least one other client holds an OtherRead lease.

State D-3 differs from state D-2 in the third condition enumerated in the previous paragraph. There is at least one client Z that holds a SelfWrite lease. As far as the server knows, it might be the case that client Z's SelfValue is True, in which case recalling its SelfWrite lease could cause client Z to become a shield for client Y. (Note that the server does not know client Z's SelfValue, because the client holds a SelfWrite lease, which permits client Z to change its SelfValue without informing the server.) In this state, the

server is not only enabled to recall client Y's OtherRead lease as it was in state D-2, but it is also enabled to recall client Z's SelfWrite lease.

When client Z returns its SelfWrite lease, it informs the server of its associated SelfValue. If client Z's SelfValue is True, then client Z acts as a shield for every client except client Z itself; thus, the state will change to D-1 if client Z ignores or is shielded from client X's SelfValue, and it will remain in state D-3 otherwise. If client Z's SelfValue is False, then the state will change to D-2 if no other clients hold SelfWrite leases, and the state will remain in D-3 if at least one other client holds a SelfWrite lease.

## *4.8 Policy for processing OtherRead requests*

The state diagrams described above permit flexibility to a server, allowing it to make policy-based decisions about which action to perform in each state. In Farsite, we have implemented a specific set of policies that we believe to be reasonable. Figs. E and F illustrate the procedures performed in Farsite for responding to requests for OtherRead and SelfWrite leases, respectively.

Fig. E shows a flow diagram illustrating the steps a server follows when processing a request from client X for an OtherRead lease. In step E-1, if client X already has the lease, the server is done. If not, in step E-2, the server determines whether it is in state C-1; if it is, then in step E-3, the server grants client X an OtherRead lease with an OtherValue of True. If the server is not in state C-1, then in step E-4, the server determines whether it is in state C-2; if it is, then in step E-5, the server grants client X an OtherRead lease with an OtherValue of False. If the server is not in state C-2, then (by process of elimination) it must be in state C-3, so step E-6 recalls a SelfWrite lease from at least one other client that holds the SelfWrite lease. In our implementation, we recall all of the SelfWrite leases at this point, but one could be more conservative and recall only a subset of them, in the hope that one of the SelfValues in the subset is True, thereby obviating the recall of additional SelfWrite leases. After these one or more recalls are issued, the server waits until the leases are returned, and then it performs the steps of this flow diagram again.

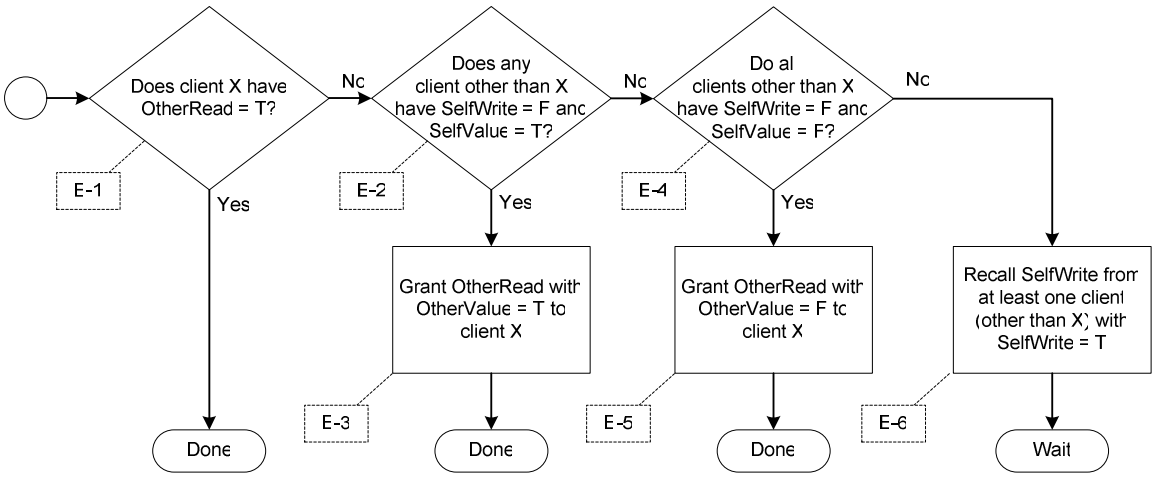


*Figure E. The steps a server follows when processing a request for an OtherRead lease*

## 4.9 Policy for processing SelfWrite requests

Fig. D showed that, in state D-3, a server has two paths by which it can transit to state D-1: It can recall an OtherRead lease from a client that does not ignore Client X's SelfValue, or it can recall a SelfWrite lease from a client in an attempt to cause that client to become a shield for others. The former action is also enabled in state D-2, but the latter action is not. As a matter of policy, when processing a SelfWrite request, Farsite does not perform the latter action of recalling SelfWrite from other clients. Therefore, the distinction between state D-2 and D-3 is immaterial to the actual SelfWrite request processing in Farsite. For this reason, the following description does not make reference to the states of Fig. D.

Fig. F shows a flow diagram illustrating the steps a server follows when processing a request from client X for a SelfWrite lease. In step F-1, if client X already has the lease, the server is done. If not, step F-2 determine how many shields there are for client X's SelfValue. There are three cases: In the first case, there are at least two shields; thus, the two (or more) shielding clients will shield each other, and they will also shield every other client, so control passes to step F-5, which grants SelfWrite to client X. In the second case, there is exactly one shield, and step F-4 determines whether the shielding client ignores client X's SelfValue. If the shielding client does ignore client X's SelfValue, then step F-5 grants SelfWrite to client X; otherwise, step F-7 recalls the OtherRead lease from the shielding client. In the third and final case, there are no shields, and step F-3 determines whether there are any other clients that do not ignore client X's SelfValue. If there are no such clients, then step F-8 grants SelfWrite to client X; otherwise, step F-6 recalls the OtherRead lease from all such clients. After issuing the recalls of step F-6 or step F-7, the server waits until the leases are returned, and then it performs the steps of this flow diagram again.

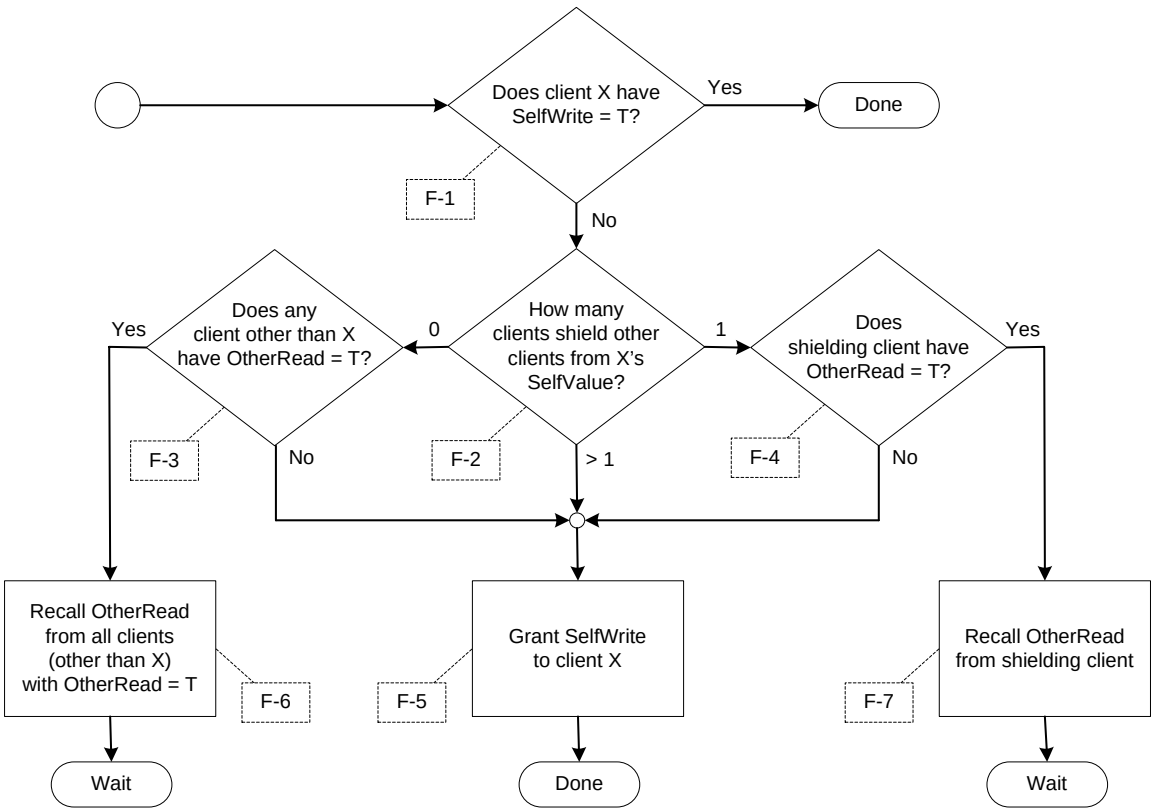


*Figure F. The steps a server follows when processing a request for a SelfWrite lease*

# 5 Summary

A black-box lease maintains values that are particular to distinct clients. These values are partitioned among clients by the lease mechanism itself, rather than in an ad hoc manner on top of a shared-value lease mechanism. A black-box lease presents to each client an aggregation of other clients' values.

The advantage gained by so partitioning the values is that clients share only information that is required to be shared. Information hiding improves concurrency and security.

# 6 References

1. A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated available and reliable storage for incompletely trusted environments. In 5th Symposium on Operating Systems Design and Implementation (OSDI), December 2002.

2. C. Gray and D. Cheriton. Leases: An efficient faulttolerant mechanism for distributed file cache consistency. In Proc. 12th Symposium on Operating Systems Principles (SOSP), pages 202--210, December 1989.