Token Module for OSF/1 AD Version 2

David Noveck

OSF Research Institute
dnoveck@osf.org

July 1, 1994

Abstract

OSF/1 AD Version 2 implements a token module to allow multiple servers to coordinate their activities. The interface to the token module allows various components of the AD server to use tokens to implement mutual exclusion and provide coherent read/write data caching. The design uses callouts to obtain tokens and return tokens so that all communication among servers is controlled by the specific AD component dealing with a given object. This enables piggybacking of token operations on object-specific messages, allowing communication to be minimized. This document describes the token module with particular emphasis on its externally visible interface and data structures.

1 Introduction

The token module[1] for OSF/1 AD Version 2 provides services to allow multiple servers, typically located on different nodes, to coordinate their activities. Facilities for mutual exclusion and to allow coordination of multiple distributed caches are provided.

This document describes the token module with particular emphasis on its externally visible interface and data structures.

Although token services may be used by many system modules, the original motivation for and the first user of the token module is ADFS. Discussion of use of the token module will be illustrated with examples from ADFS.

This document describes the token module as currently implemented in OSF/1 AD Version 2. The first static build which has the exact functionality described here is ad2b17.

The token server module in ad2b17 is a limited-function implementation which only supports eight servers and contains certain other limitations noted below. This is scheduled to be replaced

[1]This work was supported in part by the Advanced Research Projects Agency (ARPA) and the Air Force Materiel Command (AFMC).

with a full-function implementation in 3Q94. The token client module in ad2b17 is fully functional and no major revision is anticipated.

2 Basic Architecture

Tokens represent the right to perform certain actions to modify or cache certain data. The relationship between tokens and the actions that may be performed when they are held is up to the application using the token module to define. Each server module using token services establishes its own conventions which specify what tokens must be held to perform a given action. Code conforming to such conventions makes sure that the required tokens are held at the required times. The token module, by implementing its own defined semantics regarding what tokens may be held simultaneously by multiple servers, will prevent inappropriate actions on multiple servers. It does this with no knowledge of the application's semantics for its tokens.

Tokens are associated with *objects.* For any given object, one server is responsible for assigning tokens, deciding to which servers they should be granted and from which servers they should be taken away. The *token server module*, which contains code to make these decisions may not be present within every OSF/1 AD server. The *token client module* will be present within each OSF/1 AD server so that all servers may use token services even if they do not have the capability of assigning tokens. It is up to each application to decide where the token server for its objects will reside. Generally this will be where the master copy of the token-controlled object resides.

The token server and token client modules within different servers do not interact directly. Instead, they do their communication with the appropriate application code within their own server. The application code within each server is responsible for actually performing the necessary communications. This architecture allows different application to customize their communication for their specific needs, combining transfer of tokens with messages used to perform related operations.

3 Token Structure

Each token-controlled object has associated with it a set of tokens. Each token has a *class* and a *level.* Tokens for different objects and for different classes within the same object are independent. They may be assigned to different servers at the same time. Tokens of the same class for the same object always conflict. Only one level may be outstanding at a given time.

There are three levels currently defined: *read, write,* and *controlled modification.* Read-level tokens may be held by multiple servers at the same time. However, no tokens of other levels of the same class may be held while read-level tokens are outstanding. Write-level tokens provide exclusive access. When a write-level token of a given class is outstanding, no other token of that class may be assigned to any other server. This provides a cross-server mutual-exclusion mechanism. If an application modifies cached data only when possessing the write-level token of a given class and maintains a read-only cache of such data only when it continues to posses the corresponding read-level token of the same class, a distributed, coherent caching facility may be built.

The third level of token, controlled modification, is designed to support multiple updates of shared data when the application has a procedure, based on the particular semantics of the data involved to resolve multiple simultaneous updates. An example is the handling of access-time when there are multiple simultaneous readers of a file. Each reader has a read-level token for the token class associated with the file data allowing multiple readers to cache data and satisfy user read requests in parallel. Because all such readers must modify the access time, however, it requires special handling in terms of tokens. The solution used in ADFS is to provide a separate token class for the access time and for servers satisfying a read system call to get the controlled-modify level of that token. The token server allows multiple such tokens to be outstanding at the same time, allowing multiple reads to proceed in parallel. Multiple simultaneous updates can be resolved by only updating the master access-time in the server when a client access time is later than that already held. Since read-level and controlled-modify-level tokens conflict with one another, a stat from another server would require that the readers give up their access-time tokens. Multiple readers and multiple stat'ers could each proceed in parallel but there could be no mixture of the two groups.

The conflicts recognized by the token server are of only two types: conflict between two tokens of different levels for the same class on the same object and between two write-level tokens for the same class on the same object. No other conflicts are recognized. If an application, wants two operations to conflict, it effects this by making sure that the tokens required for the two operations involve some conflict. For example, read-write atomicity is ensured by the fact that read requires the read-level and write the write-level of the token class which ADFS associates the file data for a given vnode. The conflict between stat and a cached update of the access-time on another server is ensured by requiring that a reader have, in addition to the read-level data token, a modify-level token for the access-time while a stat must have read-level tokens for all of the data items which stat returns. By breaking out certain file attributes and assigning them to separate tokens as was done for the access time, greater parallelism may be obtained while providing the required semantics. In the case of stat, by getting the read-level access-time token, it causes the corresponding modify-level token to be taken away from server which have recently done a read. Note that when tokens are taken away the updated data must be sent with the returned token or sent to a central repository before it is considered returned. How this is done will be discussed below.

Because many operations will require multiple tokens in order to make appropriate conflict visible to the token module, most operations will require sets of tokens. The interface to the token module is in terms of token sets. The define TSMAKE (in token.h) is provided to construct singleton token sets by specifying a class and level with multi-element token sets constructed by or-ing together these singleton token sets. Token classes are defined by the application. The token module defines the number of possible token classes which is currently nine.

To avoid deadlocks, tokens must be obtained in accord with a hierarchy. The rule which must be respected is to always obtain tokens for a given object in order of ascending class number. Since the application defines these numbers, it can define a hierarchy appropriate to its needs.

4 Interface to the Token Module

4.1 Acquiring Tokens

Application code on a server must *acquire* a token set which is necessary for a given operation. Acquisition of a token consists of up to two operations. First, the token must be *obtained* if it is not currently possessed by the current server. Obtaining the token gets it from the token server. Once it has been obtained, it is cached by its server until the server requests that it be returned or until the local server needs a conflicting token, requiring that this token be given back to the token server. The application may also decide to return the token without a token server request but generally tokens once obtained will be retained by the local server for as long as possible to minimize communications.

When an acquire operation encounters a token which the server does not posses, it requests it by making a callout to the application code. This code can then obtain the token by sending an RPC to the application code on the server providing the token server function for the current object. This code can then get the required token and send it back to the client-side server together with any application-related data associated with the token.

On the remote server, application-specific code requests the token on behalf of the originating server. The token server keeps track of all tokens granted to individual servers and determines which tokens currently granted to a server must be taken away because of conflicts. (See section 4.2). Figure 1 shows the flow of control for an acquire operation which needs to go remote to obtain the specified token. This figure shows the communication between client-side and server side done

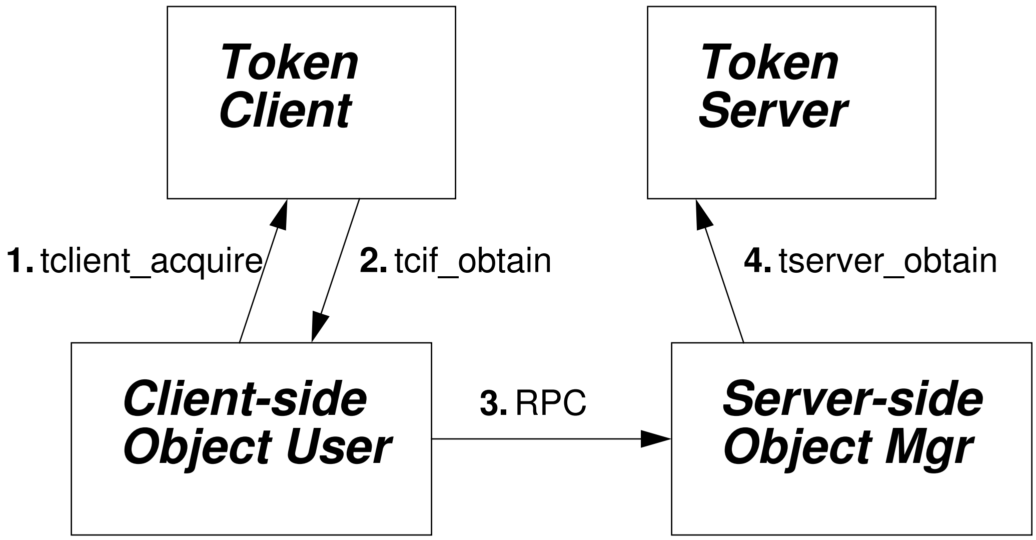

Figure 1. Token acquire with obtain

using an RPC, which is the way in which ADFS does it. Other applications could communicate in other ways.

When a tclient_acquire operation finds that a token is not present on the server, it requests that it be obtained by doing a callout to an application-specific tcif_obtain routine. The tcif_obtain routine will then generally send an RPC to corresponding code on the server side which will then obtain the token on behalf of the requesting client-side server and return that token together with other application-specific information in the response to the RPC. The tcif_obtain routine can then return to tclient_acquire which returns to the original caller.

Once a token is possessed by the server, an acquire operation causes it to be *held.* While a token is being held, it cannot be taken away from the current server. The thread doing the acquire is therefore assured that whatever operation which possession of the token permits can be done with the assurance of continuous possession of the token. When the thread no longer requires this assurance, it does a *release* to undo the hold. Once there are no active holds in effect, a token will generally enter a *cached* state in which it can be taken away from the current server if necessary. Many threads may hold the same token on the same server. No within-server exclusion mechanism is implemented by the token module. Any required exclusion between threads within the server must be implemented by the application using locks.

4.2 Revoking Tokens

When a token must be taken away from a given server, it is *revoked.* A revocation request is made by the token server and communicated to the application on the token client server. The application may then do a revoke operation on the specified tokens. The revoke may not take place immediately because of the possibility that the tokens to be revoked are currently held. At a time at which the tokens being revoked are not held, a call-out to the application is made. The callout routine can then *return* the token to the token server. He may return the token by sending a message to the application code on the server providing the token server function. This message can contain application-related data associated with the token as well as an indication of the tokens to be returned. This code can than actually give the token back to the token server module on behalf of the client-side server. Figure 2 shows the flow of control for a revoke operation. Note that the revoke operation was probably prompted by an obtain of the same or a conflicting token by another server. The figure does not show the action that prompted the revoke.

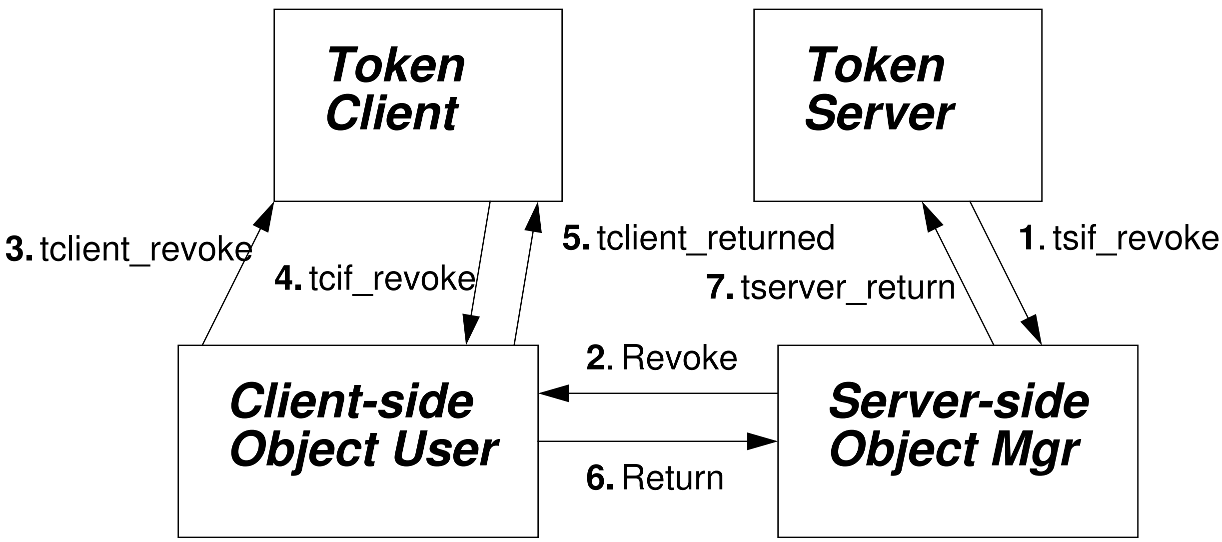

Figure 2. Token revocation and return

In the figure, the token server calls the application-specific tsif_revoke routine which sends a revoke message to the application-specific code on the client side. This code begins the revoke processing by doing a tclient_revoke call. When the specified token(s) are no longer held, the token client module does tcif_revoke callout to the application which then does a tclient_returned to notify the token client module of the new status of the tokens and sends a message returning those tokens to the server-side application code which then does a tserver_return to notify the token server module that the client no longer has the tokens. The revoke and return are illustrated here as two one-way messages which is the way that ADFS implements this. Using an RPC is possible but would require the application to deal with the possibility of more than one tcif_revoke call being associated with a single revoke RPC.

When revocation callout occurs, the application need not return the token immediately, as illustrated in figure. Although it is obliged to do so in a *reasonable* amount of time, or refuse the revocation (see below), he may defer the two required actions of calling tclient_returned and tserver_return. In this case, these action, instead of being done in the tcif_revoke routine, might be done by the application routine calling tclient_revoke or even be done at a later point. If the token return is deferred until some data structure state is true, any action preventing that state from being the case, will effectively act as a hold on the associated token. This technique can be valuable in providing a very fast mechanism to hold tokens that bypasses the normal overhead of the token acquire code and in implementing token-base solutions when the current interfaces would normally not provide them.

An example of the use of this technique is the handling of existence tokens within the ADFS client code. Within ADFS an existence token on a file guarantees the continued validity of the handle for the ADFS file. This handle is used to address the specific file in all communication between ADFS client and ADFS server. A client-side ADFS vnode without assured validity of the handle is useless. Therefore, when a client ADFS vnode is VREF'ed, assuring continued validity of the vnode, we would also like to be able to rely on the continued validity of the associated handle. This is done by having the tcif_revoke routine only return existence tokens if there are no current references for the associated vnode. If the tcif_revoke routine encounters a current reference, it sets a flag which will cause all tokens to be returned at VOP_INACTIVE time. In this way, a VREF is made the functional equivalent of a hold on the existence token, assuring the continued validity of the file handle whenever the system's generic file handling code may assume the continued validity of the associated vnode.

When a token is revoked, the application or the token client module on its behalf may refuse the revocation. The decision of what token revocations may be refused and under what circumstances is up to the application-specific code with which the token module interacts. The following examples illustrate uses of revocation refusal by ADFS.

- Revocation of existence tokens is refused when a file is open. The server-side code which is used to remove a file tries to acquire the write-level token of the same class as the existence token. To get this token, all outstanding existence tokens must be revoked. If any files are open the revocation will be refused. The server-side code deal with the refusal by leaving the removal of the file data to VOP_INACTIVE time. The client-side code which did the refusal takes a hard reference on the file and sets flags so that when the file is closed that hard reference together with all tokens will be returned, triggering removal of the file data at VOP_INACTIVE time when all clients with open files give up their references.

- Revocation of tokens associated with both open-for-write and execution are refused whenever the other token is held. These two tokens are assigned different levels (read and controlled-modify) of the same class. This allows there to be multiple servers who have a file opened for write or multiple servers which have it exec'ed but not both. An attempt to open for write (for example) needs to get the controlled-modify token which needs to revoke all the read-level tokens associated with outstanding exec's. If any of these are actively held, indicating a current exec, the revocation is refused and the token cannot be obtained, causing an error on the open. A similar path causes an exec to fail when a file is open for write. This form of revocation is implemented by the *auto-refuse* feature of the token module which allows token revocations to be automatically refused when the corresponding token is held.

5 Client-side Data Structures

5.1 Token Client Object

Each token client object is represented by a *tcinfo*. The tcinfo is presented to the token client module when requesting services of the client-side token module. Allocating space for the tcinfo is the responsibility of the application who will generally embed the tcinfo within its own structure providing the client-side view of the object in question. The tcinfo should be considered an opaque structure. Although the header files included by the application contain the definition of the tcinfo for sizing purposes, the application should refrain from accessing any fields within the tcinfo.

The tcinfo contains within it a void * which is intended to designate the application's client-side object. This is treated as opaque to the token module and is presented as a parameter to all of the indirect routines called by the token module.

5.2 Token Client Interface

The tcintfc structure is used to allow the customization of the interface between the token module and the application. There should be only one or a few tcintfc's for a given application. The tcintfc to be associated with a given token client object is established when the object is built and never changed thereafter. The tcintfc is a fully public structure and the application is free to it set up in any way it chooses. It should never be changed while in use, however.

Currently the tcintfc contains the following items:

- tcif_auto_refuse, the set of tokens which will be refused if a revocation request is received when the token is held.
- tcif_revoke, the routine to call when a token which has been revoked is no longer held.
- tcif_obtain, the routine to call when an acquire operation finds that it needs to obtain a requested token.

A special tcintfc is constructed to allow token-client services to be provided on the same server as the token-server module. Special callouts allow tokens to be obtained directly from the token server and to be returned immediately when not held with no caching.

6 Client-side Interface Routines

Interface routines used to interface with the token client code are described below. Both routines exported by the token client module, beginning with tclient_, and callout routines, beginning with tcif_, are listed.

6.1 Setting up and Destroying Token Objects

```
void tclient_initialize(void)
```

Does any necessary module initialization. This routine is called from ad_init if the token client module is configured. No token client routines may be called until tclient_initialize is executed. It is assumed that this routine is invoked in a single-thread environment so no locking is done.

```
void tclient_setup(
    tcinfo_t * info_to_set_up,
    void * client_side_object_pointer,
    tcintfc_t * interface_to_use,
    tset_t tokens_gotten_already)
```

Used to set up a tcinfo for a new token-controlled object on the client side. Allocating the storage which the tcinfo will occupy is the responsibility of the caller. Generally, the tcinfo will be declared as a substructure of the data structure which is the client-side representation of the object in question. The client_side_object_pointer allows the address of this object to be specified so that the token client module can make it available to the user when do a callout. Callouts to be used are specified in the tcintfc provided. The caller may specify a set of tokens which are to be considered as already having been obtained. In ADFS, the response to a lookup request will generally contain an existence token which confers the right to cache the handle for the file and attribute and name tokens as well. Whenever tclient_setup is called for a file, tokens_gotten_already will always contain the existence token and generally the name and attribute tokens as well. Without the existence token, the continued validity of the ADFS vnode in which the tcinfo lives cannot be assumed.

```
void tclient_prepare_destroy(
    tcinfo_t * tcinfo_which_will_be_going_away,
    tset_t * tokens_to_be_returned)
```

Used to begin the process of destroying a token-controlled object. The caller must make sure that no tokens are held (by tclient_acquire) or otherwise unavailable (tclient_obtaining, tclient_returning, tclient_revoke). In ADFS, this is done by making sure that there are no references on the vnode for the file in question. All token operations are done with vnode references held. The caller receives the set of tokens which the object currently has. By calling tclient_prepare_destroy, it undertakes to return these to the token server. Generally this will be done by sending a message to the server side which will return the tokens and also release any references on the server-managed object. In ADFS, since the existence token acts as a soft reference, these operations are unified conceptually.

Once a tclient_prepare_destroy has been done, the caller must not call any other token client operations on the specified tcinfo, except to do a tclient_destroy. The token client module will not do any callouts on the specified tcinfo.

```
void tclient_destroy(
    tcinfo_t * tcinfo_going_away)
```

Used to terminate use of a given tcinfo. Once this routine is called, the token client module will make no further reference to the tcinfo storage. It is valid to call tclient_destroy immediately after calling tclient_prepare_destroy but it may be better to wait until the token client user is done using the data structure in which the tcinfo is located and has taken his own steps to assure that the data structure is not referenced again.

6.2 Providing a Client Interface on the Server

Code on the server where the token server for a given object is located may interact with the token server module directly by using the tserver_ interfaces. (See Section 8). In some cases, however, it may more convenient to use the tclient_ interfaces even on the token server server. This allows use of the hold-counting feature of tclient_acquire for example. On servers compiled with both the TOKEN_SERVER and TOKEN_CLIENT options, the following routines are used to set things up so that this functionality can be provided.

```
void tclient_local_interface(
    tcintfc_t * tc_interface_to_set_up)
```

Sets up a special tcintfc to be used when setting up tcinfo's for local use. The caller should have previously filled in all the fields except the callouts, tcif_obtain and tcif_revoke. These callouts are set up to point to special routines which obtain tokens locally by requesting them from the token server module and which also provide that tokens will not normally be retained when not held by the token client module but will be returned immediately to the token server.

```
void tclient_local_setup(
    tcinfo_t * tcinfo_to_set_up,
    tsinfo_t * corresponding_tsinfo,
    tcintfc_t * tcintfc_set_up_by_tclient_local_interface,
    tset_t tokens_gotten_already)
```

Used like tclient_setup when we are setting up a token client interface on the same server as the token server module. Instead of application-specific callouts being used, a special set is used to interface token client and token server functionality. The tsinfo address specified is saved as the client object pointer. This is then used in the tcif_obtain routine to do the appropriate tserver_obtain directly. Tokens obtained are immediately revoked so that they will be returned to the server when no longer held. The tcintfc passed must be one set up by tclient_local_interface.

6.3 Acquiring and Releasing Tokens

```
void tclient_acquire(
    tcinfo_t * current_tcinfo,
    tset_t tokens_wanted)
```

```
tset_t tclient_acquire_conditional(
    tcinfo_t * current_tcinfo,
    tset_t tokens_wanted)
```

Used to hold a specified set of tokens, obtaining then from the token server (through tcif_obtain) if necessary. When a token is held, it may not be revoked. Revokes which occur between a tclient_acquire and the corresponding tclient_release, are held pending until all acquire's have been released. Multiple threads may hold the same tokens simultaneously, even write-level tokens which obey a single-writer discipline among multiple servers. The token module maintains a count of current holds. If mutual-exclusion is required within a server, it is up to the server to use locks or some equivalent mechanism.

If the current thread has already acquired a set of tokens, the current acquire must only be for tokens whose class numbers are higher than any in the set of tokens already acquired by this thread. Failure to obey this rule may result in deadlock.

To call tclient_acquire, the caller must be certain that none of the tokens to be acquired can ever be refused. If some of these tokens cannot be obtained because of refusal of a revocation, the token client module will panic. When there is a possibility that one or more of the tokens to be acquired cannot be obtained successfully, tclient_acquire_conditional should be used.

When we currently have a token of the same class as one of the tokens requested but of a different level, processing proceeds as follows:

- If the token level which we have dominates the one which is requested, then the request is immediately granted. Write-level tokens are considered to dominate both of the other two levels, neither of which dominate the other. Thus, if a read-level token is requested while we have the corresponding write-level token, the acquire is granted immediately, whether the write-level token is held or not.

- Otherwise, if the conflicting token is specified in the tcif_auto_refuse set and is held, the request is refused immediately. ADFS uses this feature to arbitrate between exec and open-for-write. In this way, we can avoid a remote transaction when the information necessary to deny the request is available locally.

- If we have a conflicting token and auto-refuse is not done, then the conflicting must be returned and the requested token obtained. If the conflicting token is held, these operations (done through tcif_obtain) are deferred until the conflicting token is no longer held.

```
void (*tcif_obtain)(
    tcinfo_t * tcinfo_for_current_object,
    void * current_object_pointer,
    tset_t tokens_to_obtain,
    tset_t tokens_to_return,
    tset_t * tokens_server_already_had,
    tset_t * tokens_which_were_refused)
```

The specified routine is called by the token client module when an attempt is made to acquire a token which the token client module does not have on this server. It is used to allow the user of the token client module to participate in obtaining tokens. This allows it to combine the transfer of data associated with tokens with the tokens themselves. The token module provides the address of both the tcinfo and the client object data structure. Although normally only the client object pointer would be required, providing the tcinfo is necessary for the routines which provide token client functionality on the same server as the token server module.

The token module specifies a set of tokens to obtain and a set, possibly null, to return. This is because it may be necessary to return tokens in order to obtain conflicting tokens. The tcif_obtain routine returns an indication of tokens that it was unable to obtain or to return.

First, tokens_server_already_had is set to a subset, possibly null, of the tokens to be obtained. This is to deal with races in which multiple requests are made of the server-side code to obtain a token and one finds that the token which it was requested to obtain has already been marked as obtained on behalf of the current client server. If a token is always obtained using tcif_obtain, and/or tclient_obtaining and tclient_obtained, such races cannot occur because the token client module will synchronize multiple attempts to obtain a token. If the specified token is obtained on the server without notifying the token client, as can happen, for instance, when doing ADFS lookup (it would be impossible for the token client to be notified since at the point the token is obtained on the server, he has no knowledge of the object in question), such races can occur. When an obtain fails in this way, the token client expects that a tclient_gotten (see below) has been or will be done on the token in question and when this happens those waiting for he token will be able to proceed.

Also a set of tokens refused is returned. The most common case of refusal is for tokens requested to be obtained. This will result from the refusal of a revocation on another client server. This refusal will be propagated to the caller of tserver_obtain who will send it back to the tcif_obtain routine which will return the information to the token client module who will return it to the caller of tclient_acquire_conditional.

It is also possible for the return of tokens to be refused. Just as the revocation of tokens may be refused, the tcif_obtain routine may also refuse to return tokens when the application would prefer that this be refused. Generally, this will be under the same conditions in which a revocation for the same token, coming on behalf of a request by another client server, would be refused. When the tcif_obtain routine refuses to return a token, it must also refuse to obtain the conflicting token which prompted the return.

```
void tclient_release(
    tcinfo_t * current_tcinfo,
    tset_t tokens_to_release)
```

Releases tokens held by tclient_acquire. If any of the tokens have been revoked while held or if there is pending acquire of a conflicting token or there is a pending tclient_obtaining or tclient_returning call, one of these actions will be done if this is the last hold on that particular token.

```
tset_t tclient_hold(
    tcinfo_t * current_tcinfo,
    tset_t tokens_to_hold)
```

Holds tokens which are currently being cached locally are held. The value returned is the set of tokens on which a hold was done. If the token is not present within the current server or is in the process or being revoked or returned, nothing is done. If a token was already in the process of being obtained, tclient_hold will wait for it and then do the hold. The caller should do a tclient_release to undo the hold on any of the tokens in the set returned by tclient_hold.

6.4 Token Revocation and Return

Tokens may be returned as a result of a revocation request prompted by the token server or the token client user may initiate return of a token when doing so would avoid a likely revocation transaction.

```
void tclient_revoke(
    tcinfo_t * current_tcinfo,
    tset_t tokens_to_revoke,
    tset_t * tokens_refused)
```

Initiates a revoke on the specified tokens. Generally this will be prompted by a tsif_revoke call on the token server. Tokens which are not held will immediately be made the subject of tcif_revoke calls. The tcif_revoke call will be done for others as soon as they are no longer held.

If any of the tokens specified are specified in tcif_auto_refuse of the associated tcintfc structure and are currently held, they will not be revoked. Instead, the set of such tokens will be returned in tokens_refused. This feature is used in ADFS to handle the conflict between exec and open-for-write. A request for the open-for-write token, if it encounters a held, conflicting, exec token will result in refusal. The open will be rejected with ETXTBUSY.

```
void (*tcif_revoke)(
    tcinfo_t * tcinfo_for_current_object,
    void * current_object_pointer,
    tset_t tokens_to_revoke)
```

This routine is called when the token client module believes that a revoke should be honored and the token returned to the token server module. By giving the application control at this point, it is given a chance to send modified data back before returning the token or if the data is small to send it back with the token to be returned. Also, if revocation invalidates cached data, the actual invalidation is appropriately done in the tcif_revoke routine.

A further function of the tcif_revoke routine is to determine when and whether the revocation should be honored. Even though the token client module is ready for the token to be returned, the application may defer the return, or refuse it.

The return may be deferred if there are conditions present that the tcif_revoke wants to treat as equivalent to a token hold. In ADFS, a reference to the associated vnode is treated as equivalent to a hold on the existence token for that file. Since an existence token assures the continued validity of the vnode, and doing a VREF is designed for that same purpose in a local environment, it is desirable that simply doing the VREF has the same effect as acquiring the existence token. Deferring return in the tcif_revoke routine causes VREF to have precisely this desirable effect without changing the generic file system code that does VREF's. Note that we get the same effect more efficiently, for this special case, than by doing a tclient_acquire.

Revocation may be refused in some cases if that is appropriate to the application semantics. Deferral is inappropriate when a situation which prevents return can go on for an unbounded time. In ADFS, the conflict between file removal and open illustrates this. Removal attempts to get the write-level token corresponding to the existence token. Normally, the revocation should remain pending because of the VREF associated with the open. This would cause the remove to remain pending indefinitely. To prevent this, the tcif_revoke routine simply refuses the revocation if the file is open. The vnode is marked so that last close will cause a return of the existence token with a hard reference taken on the remote file. This will cause VOP_INACTIVE on the server side and removal of the disk when all clients give up their references.

```
void tclient_returning(
    tcinfo_t * current_tcinfo,
    tset_t wants_to_return,
    tset_t * should_return,
    boolean_t wait_to_resolve)
```

Used when the user wants to return a token to the token server without a revocation occurring. Returning a token in advance of revocation is desirable when the application knows that it will not be needed in the near future or it is likely to be revoked in the future. For example, ADFS does all set-attribute operations on the server which manages the associated on-disk file system. Setting attributes will therefore always require that this server get the write-level file attribute token, causing any attribute token on the client server to be revoked. Therefore, it is appropriate for the client-side set-attribute code to undertake, by using tclient_returning, to return this attribute token if we currently have it.

The caller specifies a set of tokens that he wants to return. A subset of these, possibly proper, is returned to him. A token will not be in the returned set if this server does not currently have it. Other reasons that a token may not be included in this set are controlled by the wait_to_resolve parameter. When this parameter is FALSE, tokens which are held or in the process of being returned or obtained will simply be omitted from the set of tokens the caller is given back to return. When wait_to_resolve is TRUE, tclient_returning will wait for any of these situations to pass before returning.

Tokens given back to the caller for him to return are in a locked state and cannot be used until the caller unlocks them by doing a tclient_returned. This call will generally be made after doing an RPC to the server side in which the tokens are returned. Doing it earlier may result in races if a revoke is sent by the server and received on the client after the tclient_returned is done in which case we will have a revoke for a token which we do not think we have.

```
void tclient_returned(
    tcinfo_t * current_tcinfo,
    tset_t tokens_returned,
    tset_t tokens_refused)
```

Used to signal the disposition of tokens that the caller has an obligation to return. There are two situations in which such an obligation will arise:

- As a result of a tcif_revoke call from the token client module.
- As a result of a tclient_returning call by the user.

The caller specifies a set of tokens which have been returned. These tokens are considered no longer to exist on this server. If they are needed they will need to be obtained anew. A separate set of tokens whose return has been refused is also specified. These tokens are considered retained on this server. Threads waiting to acquire them will proceed and be allowed to hold them. In both cases the token classes associated with these tokens are unlocked so that waiting obtains can proceed.

The caller need not dispose of his obligations to call tclient_returned in a single call. He may make individual calls for any subset of the tokens if he so desires.

The time at which it is appropriate to call tclient_returned differs for the two situations. In the case of a revoke-induced return, tclient_returned may be called immediately, in the tcif_revoke routine. This is valid because the token server having initiated the operation, has enough information to prevent races between subsequent operations on the same token which the revoke is being processed. The token return can be done by a one-way message.

In the case of a client-initiated return, tclient_returned cannot be called until we know that the server that no operation on the same token can be done before receiving notification that the return has happened. This can be done through message sequencing but the simplest way of getting this assurance is by doing the return as part of an RPC and only doing tclient_returned when that RPC returns.

6.5 Obtaining Tokens Directly

Facilities are provided to allow the user to obtain and hold tokens directly without calling tclient_acquire and having the tcif_obtain routine called. This more flexible mechanism allows applications with special needs to customize the transmission patterns of token requests for specific situations. The routines tclient_obtaining and tclient_obtained, exported to the applications so they may customize their own transmissions are the same ones used by tcient_acquire_conditional to implement the more general token acquisition facility.

```
void tclient_obtaining(
    tcinfo_t * current_tcinfo,
    tset_t tokens_desired,
    tset_t * should_obtain_these,
    tset_t * should_return_these,
    tset_t * these_were_refused,
    tset_t * get_these_later)
```

Used by code which wants to get tokens as part of another operation. For example, in ADFS, the RPC which preallocates disk for a write operation would want to get the write-level data token. Effecting this kind of piggybacking is difficult using the tcif_obtain paradigm because you would need to propagate such information as how much disk to preallocate through a number of levels from the application code through the token client module to the tcif_obtain routine.

By using tclient_obtaining, this problem is eliminated. The caller has all of his application-specific context when issuing his RPC. The caller specifies what tokens he would like to have and the token client module responds by returning to him four separate token sets him to act on. Of these all but the second (tokens to return) will be subsets of the tokens that he has indicated a desire to obtain.

- Tokens for him to obtain as part of his RPC. The caller should attempt to obtain each of these from the token server and then call tclient_obtained to indicate the results of the attempt. Until this is done, the associated token classes are locked. Once the call to tclient_obtained is done, these tokens are held just as they would have been held if a tclient_acquire was done.

- Tokens for him to return because they conflict with tokens he will be obtaining. The caller should return these tokens on the token server before trying to obtain the conflicting tokens. After returning from the RPC, tclient_returned should be called for these tokens. There is currently no exported interface to deal with an application-specific refusal to return these tokens as there is in the tcif_revoke case. If there is a need for it, it will be added.

- Tokens which cannot be obtained because of the auto-refuse feature. The application should treat these in the same way as it would those whose refusal was signaled by the token server.

- Token which need to be gotten later. Tokens in this set cannot, because of token hierarchy constraints, be held at this point. For example, if tclient_obtaining, going in order of ascending token class number, finds first a token which is not present and needs to be obtained and then later finds one which is present (normally only requiring a hold), this later token cannot be held because it must be revocable while the first one is being obtained. Such tokens must be obtained later after the first token (it could be a set) is obtained from the server. The caller has a choice of getting these tokens by doing a tclient_acquire or using tclient_obtaining and its own RPC's for the secondary token obtain as well.

Tokens that were specified as those that the caller desired to obtain but are not in any of these sets can be considered successfully acquired. They only need to be tclient_release'd when the caller is done with them.

```
void tclient_borrowing(
    tcinfo_t * current_tcinfo,
    tset_t tokens_being_borrowed)
```

Used to mark tokens being obtained as having a limited lifetime. Such tokens will not be cached but are returned (via a tcif_revoke call) once they are no longer held. Such tokens are different from tokens marked revoked in that subsequent acquires may be done. As long as there are threads hold the tokens, they are not returned and there is no effort made, as there would be with a revoked token, to bound this state of affairs.

Implementation of the client interface local to the token server uses tclient_borrowing to mark tokens non-cacheable and it may be useful for application-specific code as well.

```
void tclient_obtained(
    tcinfo_t * current_tcinfo,
    tset_t obtained_these,
    tset_t already_had_these,
    tset_t these_were_refused)
```

Used to signal the completion of an attempt to obtain tokens begun by tclient_obtaining. The caller specifies three disjoint token sets, whose union is a subset of the tokens that were undertaken to be obtained by calling tclient_obtaining. The caller need not dispose of all such tokens in a single call but may make as many individual calls as convenient. An individual thread should not, however, try to obtain additional tokens, whether by calling tclient_obtaining or tclient_acquire, until all of the tokens from a previous tclient_obtaining are disposed of.

```
void tclient_gotten(
    tcinfo_t * current_tcinfo,
    tset_t tokens_gotten)
```

Allows tokens spontaneously sent by server-side code to be recorded as present in this token client module. As a general rule, this routine should be avoided. Because neither the token client module does not know in advance of the attempt to acquire the token, many races, such as receiving a revoke for a token before the token itself are opened up. While the token module attempts to deal with these, and no problems are currently known, the resulting paths are more complicated and do not have the same level of assurance as those that do not involve tclient_gotten. Out experience which ADFS is that excessive use of tclient_gotten is a sign that something in the design is wrong, but that we have not been able to dispense with it entirely and meet our other goals.

7 Server-side Data Structures

Interface routines used to interface with the token server code are described below. Both routines exported by the token server module, beginning with tserver_, and callout routines, beginning with tsif_, are listed.

7.1 Token Server Object

Each token server object is represented by a *tsinfo*. The tsinfo is presented to the token client module when requesting services of the server-side token module. Allocating space for the tsinfo is the responsibility of the application who will generally embed the tsinfo within is own structure providing the server-side view of the object in question. The tsinfo should be considered an opaque structure. Although the header files included by the application contain the definition of the tsinfo for sizing purposes, the application should refrain from accessing any fields within the tsinfo.

The tsinfo contains within it a void * which is intended to designate the application's server-side object. This is treated as opaque to the token module and is presented as a parameter to all of the indirect routines called by the token module.

7.2 Token Server Interface

The tsintfc structure is used to allow the customization of the interface between the token module and the application. There should be only one or a few tsintfc's for a given application. The tsintfc to be associated with a given token server object is established when the object is built and never changed thereafter. The tsintfc is a fully public structure and the application is free to it set up in any way it chooses. It should never be changed while in use, however.

Currently the tsintfc contains the following items:

- tsif_granted, the routine to call when an asynchronous request to obtain a token is granted.
- tsif_revoke, the routine to call when a token needs to be revoked.
- tsif_reclaimed, the routine to call when a previously-initiated request to reclaim a token-controlled object completes, whether successfully or unsuccessfully.
- tsif_idle, the routine to call to notify the application that a specified token class is idle (i.e. has no tokens granted to any client). This is in response to a tserver_notify_idle call.

8 Server-side Interface Routines

Interface routines used to interface with the token server code are described below. Both routines exported by the token server module, beginning with tserver_, and callout routines, beginning with tsif_, are listed.

8.1 Setting up and Destroying Token Objects

```
void tserver_initialize(void)
```

Does any necessary module initialization. This routine is called from ad_init if the token server module is configured. No token server routines may be called until tserver_initialize is executed. It is assumed that this routine is invoked in a single-thread environment so no locking is done.

```
void tserver_setup(
    tsinfo_t * tsinfo_to_set_up,
    void * server_object_pointer,
    tsintfc_t * associated_interface)
```

Used to set up a tsinfo for a new token-controlled object on the server side. Allocating the storage which the tsinfo will occupy is the responsibility of the caller. Generally, the tsinfo will be declared as a substructure of the data structure which is the server-side representation of the object in question. The server_object_pointer allows the address of this object to be specified so that the token client module can make it available to the user when doing a callout. Callouts to be used are specified in the tsintfc provided.

```
void tserver_destroy(
    tsinfo_t * tsinfo_we_are_done_with)
```

Called when a tsinfo is no longer to be used. Once this routine is called, the token server module will make no further references to the tsinfo and the caller should make no further token server calls specifying it as the first parameter. It is the caller's responsibility to make sure that when this routine is called, there are no tokens outstanding. The simplest way of doing this is to call it from the tsif_reclaimed routine when a reclaim operation succeeds.

```
void tserver_reclaim(
    tsinfo_t tsinfo_to_reclaim
    tcmask_t classes_to_act_on)
```

Begins the process of reclaiming a tsinfo. The reclaim will not complete successfully until all outstanding tokens of the specified classes have been successfully revoked. If revocation is refused on a token, or a tserver_unreclaim is done before all tokens have been revoked, the reclaim is terminated unsuccessfully. When a reclaim operation is completed, whether successfully or not, the tsif_reclaimed routine is called.

An attempt to obtain tokens while a reclaim is in progress is an error and will result in a panic. Similarly, tserver_reclaim should not be called if there may be uncompleted tserver_obtain's in progress.

An ongoing reclaim is no barrier to tserver_return and tserver_revoke_failed calls. They are necessary to complete revocation processing to allow the reclaim to complete.

When a reclaim is in process, tserver_reclaim may not be called. This is true even if the token class masks presented for the two calls do not intersect. The reclaim is considered in process until the tsif_reclaimed routine is called.

```
void tserver_unreclaim(
    tsinfo_t tsinfo_being_reclaimed)
```

Terminates an ongoing reclaim operation. No further reclaim-induced revocations will be issued. Obtain operations will no longer be considered invalid. The tsif_reclaimed routine is called to indicate unsuccessful completion of the reclaim.

```
void (*tsif_reclaimed)(
    void * server_object_pointer,
    boolean_t success)
```

Called upon completion of a reclaim operation. If success is TRUE, the reclaim was successful and tserver_destroy may be called. The relationship between tserver_reclaim calls and subsequent calls of the tsif_reclaimed routine should be strictly one-to-one.

8.2 Obtaining Tokens

```
void tserver_obtain(
    tsinfo_t * current_tsinfo,
    tclient_t for_whom,
    tset_t token_wanted,
    tset_t * tokens_granted,
    tset_t * tokens_refused,
    tset_t * tokens_already_obtained)
```

Obtains the specified set of tokens for the specified client server. Three disjoint token sets are returned. The union of these sets is equal to the set of tokens requested.

The token server keeps track of what tokens each client has (it must do this so it knows what to revoke) and refuses to grant the same token to the same client a second time. The set of tokens which were not granted because they were already assigned to the client is returned in tokens_already_obtained. If all tokens are obtained on the client side using tclient_acquire, tclient_obtaining, and tclient_obtained, this situation (finding out that tokens are already assigned to the specified server) should never occur since the synchronization of the token client module will only allow one outstanding request to obtain for each token class. When tokens are obtained in advance, as happens in ADFS when doing a lookup, for example, this situation can occur and the application must deal with it.

Tokens which the specified server does not already posses are either granted or refused. Tokens can be refused if, there is a conflicting token currently assigned, necessitating one or more revokes, and one of those revokes is refused.

Tserver_obtain does not return until all of the tokens specified have either been assigned to the specified server or have been refused.

Tserver_obtain may not be done if a reclaim is process (see tserver_reclaim in section 8.1).

```
void tserver_obtain_async(
    tsinfo_t * current_tsinfo,
    tclient_t for_whom,
    tset_t token_wanted,
    tset_t * tokens_already_obtained)
```

Provides an asynchronous routine to obtain tokens. It returns immediately and provides for the tsif_granted routine to be called when all of the specified tokens are either granted or refused. The set of tokens already obtained is determined immediately. If all tokens are immediately available, the tsif_granted routine will be called before returning from tserver_obtain_async.

In the current limited implementation of the token server module (the one present in ad2b7), the implementation of tserver_obtain_async is not asynchronous. It simply calls tserver_obtain and then calls the tsif_granted routine.

Tserver_obtain_async may not be done if a reclaim is process (see tserver_reclaim in section 8.1).

```
void (*tsif_granted)(
    void * server_object_pointer,
    tclient_t for_whom,
    tset_t tokens_granted,
    tset_t tokens_refused)
```

Called when an obtain operation begun by tserver_obtain_async is completed. Two tokens sets are passed, indicating which of the original tokens that were requested have been granted and which refused. Tokens can be refused because of a reclaim or because a revoke is refused. Note that if a tserver_reclaim is done while an asynchronous obtain is pending, tsif_granted will be called immediately with all tokens not yet granted marked as refused.

8.3 Revoking and Returning Tokens

```
void tserver_return(
    tsinfo_t * current_tsinfo,
    tclient_t for_whom,
    tset_t tokens_returned,
    tset_t tokens_refused)
```

Called to return tokens which are currently owned by the specified server. The set of tokens returned is used to deal with revocations and client-initiated token returns. The set of tokens refused should only be set in response to revocations.

When revocation is initiated by a tsif_revoke call, all of the tokens specified for revocation must be eventually dealt with by being the subject of a tserver_return. It is not necessary that a single tsif_revoke call be dealt with by a single tserver_return. A single tsif_revoke may be dealt with by multiple tserver_return's and a single tserver_return may include tokens from multiple tsif_revoke calls.

```
void tserver_revoke_failed(
    tsinfo_t * current_tsinfo,
    tclient_t for_whom,
    tset_t tokens_revoked)
```

This invokes a revoke retry mechanism used to deal with certain races that occur when a revoke for a token is sent before the corresponding object is set up on the client side. In such cases, the tsif_revoke routine will send a handle to the client which is not recognizable yet. This situation can arise in ADFS since tokens are assigned for the new file object first made known by lookup. If a revoke should occur on any of these token and the indication of that revoke reaches the client before the response to the lookup then the handle contained in the revoke message will be unrecognizable. Keeping the revoke indication pending is one way to deal with this race but such an approach is complicated by the fact that an unrecognizable handle can also arise when the client-side vnode is reclaimed.

The solution used to deal with this situation in ADFS is to send back to the server a special indication when an unrecognizable handle is encountered. This message causes the ADFS server code to look for the specified handle on the server and if it is found, call tserver_revoke_failed. Tserver_revoke_failed will then cause a new tsif_revoke call for any of the specified tokens that have not yet been returned.

Note that this problem only arises when tokens are sent in advance of the client-side object being set up. Other races between revokes and tokens grants which result from spontaneous token assignments effected by tclient_gotten can be dealt with by the synchronization within the token client module.

```
void (*tsif_revoke)(
    void * server_object_pointer,
    tclient_t for_whom,
    tset_t tokens_to_revoke)
```

Called when it is necessary to revoke one or more tokens held by a given client. The application should send this information to the appropriate server where a tclient_revoke will be done. When the tcif_revoke routine is called, the information regarding the tokens returned can be sent here and tserver_return called.

There are several possible approaches to deal with the case of the tsif_revoke routine being called for a local client (for_whom designating the current server).

- Simply do nothing. This is appropriate when the application can ensure that token refusal cannot occur and that the tokens will eventually be returned without additional action being taken. Setting up the associated tcinfo using tclient_local_setup provides a the latter guarantee although the wait until the hold count goes to zero will not be bounded since a the weak revoke which is done will allow subsequent acquires.

- Handle this revoke like others and send a message within the same server. In applications in which the server-side code and the client-side code are very different this may be inconvenient.

- Map from the server-side object address to the corresponding local tcinfo and do the tclient_revoke directly.

8.4 Other Token Server Routines

```
int tserver_iterate(
    tsinfo_t * current_tsinfo,
    tset_t token,
    boolean_t stable,
    tsiret_t (*routine)(
        tsinfo_t * current_tsinfo,
        tclient_t who,
        void * parm),
    void * parm)
```

Allows an object manager to perform a specific action for every token client that has a specified token. The specified routine is called once for each client that has the specified token.

If the parameter stable is TRUE, the caller warrants that there should be no ongoing state change for the specified token. It may not be in the process of being obtained or revoked. Otherwise a panic will result.

The client routine may control the progress of the iteration by the value returned which is an enumeration containing the following values:

- TSIR_CONTINUE allows tserver_iterate to continue normally.
- TSIR_STOP causes tserver_iterate to stop its scan immediately.
- TSIR_RETRY indicates that the client routine is unable to perform its function currently and needs to retry. The client routine will be called again for the current client once the initial scan has been completed.

```
void tserver_notify_idle(
    tsinfo_t * current_tsinfo,
    tclass_t token_class)
```

Provides that a callout is done when no tokens of the specified class are granted to any client. The tsif_idle routine is called as soon as the last token is returned. If there are no tokens granted of this class, the callout occurs before tserver_notify_idle returns.

When idle notification has been requested, tserver_notify_idle may not be called for the same class until the tsif_idle routine has been called.

```
void (*tsif_idle)(
    (void * object,
    tclass_t token_class)
```

Indicates that the tokens class specified in an earlier tserver_notify_idle call has been made idle

Note that no lock is held when the tsif_idle routine is called. If the application needs to be assured that no such token has been granted for the duration of the call, it must so structure its own callout code.

9 Miscellaneous Routines

There are a number of miscellaneous support routines which are present both on the token client and the token server. These provide facilities to deal with token sets and to associate application data with individual tokens.

9.1 Token Set Routines

We provide a number of utility routines to deal with token sets which are implemented as masks. These will typically be used for debugging.

```
boolean_t tset_single(
    tset_t set_to_test)
```

Determines if the specified token set consists of only a single token. Returns true if this is the case.

```
boolean_t tset_unique(
    tset_t set_to_test)
```

Determines if the specified token set has no more than one token of each possible token class. Returns true if this is the case. Most token sets accepted by token client and token server routines should have this property.

```
void tset_printf(
    tset_t set_to_print)
```

Prints out a token set in terms of classes and levels. This is helpful for debugging.

9.2 Token Data Routines

The token module defines a *tdata* structure which is intended to allow one or more tokens to be associated with application-specific data. This is to ease the application's transfer of data associated with tokens when that data should be transferred to the client when a token is granted and back to the server, in modified form, when a token is returned.

```
void tdata_init(
    tdata_t * data_area_to_set_up,
    int length_reserved_for_data)
```

Initializes a token data area. When initialized it contains no tokens and no data to be associated with tokens. The caller specifies the size of the data area to be reserved. The caller is responsible for presenting an area of memory with space for the tdata proper plus the number of data units (of type tdelem_t) that may eventually be associated with it.

```
void tdata_add(
    tdata_t * data_area_to_add_to,
    tset_t token_associated_with_data,
    tdindex_t length_of_data_in_tdelem_t_units,
    idelem_t * associated_data)
```

Adds the specified token/data-area pair to the tdata being built. The specified token set should normally have only a single token within it. The resulting tdata can be sent from the token server to the token client when granting tokens or from the token client to the token server when returning tokens.

10 Token Client Internals

10.1 Tcinfo Structure

Each client-side token-controlled object is represented by a tcinfo structure. The tcinfo structure contains the following data items:

void * tci_object

This gives the address of the object represented by this tcinfo. This address is presented to the user as one of the parameters to the tcintfc-specified callouts. It is also available for debugging. It is not used otherwise since it is opaque to the token module.

tcintfc_t * tci_intfc

This gives the address of the tcintfc corresponding to the current token-controlled object. This is used to determine the appropriate callout routine and the set of auto-refuse tokens. The tcintfc can be used whenever there is information that should be stored for each object type rather than for each object.

int tci_refs

Used for reference counting when the fact that the tcinfo is being referenced cannot be represented in the hold count (See 10.2) or the state (See 10.2, 10.3).

tccword_t tci_cword[TCLASS_COUNT]

Provides a state word for each token class. The structure of the state word is given below in section 10.2.

udecl_simple_lock_data(,tci_lock)

This lock governs all the variable data in the tcinfo, which consists of tci_refs and the tci_cword array.

tdebug_t * tci_debug

This word, present only if the TCLIENT_DEBUG option is on, points to a debugging structure which is used to trace activity on a given tcinfo.

10.2 Token Client Class Word (tccword_t)

The class word contains a number of fields describing the status of tokens for a given token class. Each of the fields is defined by a mask of the bits within the fields. Macro's are defined to get and set each of these fields. The class word should never be referred to directly.

#define TCCW_MASK_HOLD 0x0000ffff

The hold count indicates how many threads have access to the specified token. When a token is possessed by the current server, the non-zero hold count prevents the token from being returned until all threads using consent (via tclient_release) to its return. Many token state transitions (See 10.3) may only occur when the hold count is zero.

#define TCCW_MASK_EXTLEV 0x00070000

The extended level represents a set of token levels which are being used at a given time. These levels will always be compatible. For example, when we have the write level for a given token class, an attempt to acquire the corresponding read-level token may proceed since the read-level token is subsumed by the exclusive write-level token. The fact that there may be a hold on both read-level and write-level tokens is encoded in an extended level for the set that includes both levels so that when the read-level token is released we can determine that the release of the read-level token may validly decrement the hold-count for the write-level token. When the hold count goes to zero., the extended level returns to the singleton set for the level that we actually posses.

This scheme will detect most, but not all, cases in which a token is released that was never acquired. Detecting all such cases would require a hold count for each level of each token class.

#define TCCW_MASK_STATE 0x00780000

The token class state represents the current state of possession for tokens of a given class. It indicates whether we posses the token, whether we are returning it, obtaining it and also what is to be done with the token when it is no longer held.

#define TCCW_MASK_OBTAIN 0x03800000

The obtain mask represents those token levels for which an attempt to obtain a token of the current class in pending. The thread which turns on the corresponding bit in the obtain mask is the one which will actually obtain the specified token.

#define TCCW_MASK_REVOKE 0x1c000000

The revoke mask represents those token levels for which a request to revoke a token of the current class is pending. A revoke mask needs to be maintained because revokes can occur (due to message races) when the corresponding token has not yet been gotten.

#define TCCW_MASK_WAITSC 0x20000000

The waitsc bit is set when a thread is waiting for a status change for the specified token class. This bit is not used when a thread is waiting for one of a set of specific common status changes for which it can be determined whether a thread is waiting using other data in the status word. These status changes are:

- When a token which is being obtained is obtained or the obtain fails and there are other threads represented in the hold count for that token.
- When a token is released, dropping the hold count to zero and a thread is waiting to return the current token or obtain a different token of the same class.
- When a token which is being returned has the return completed (a tclient_returned call) and a thread is waiting to obtain a token of the current class.
- When an obtain fails and a thread is waiting to obtain a different token of the same class.

10.3 Token Client Class States

There are thirteen token states which represent the possession status of tokens of the specified class. Each of the states is listed below together with information on the valid values for other fields and the possible state transitions.

Valid values for the hold count are represented as follows:

- States for which the hold count must always be zero are marked with *HC=0.*

- States for which the hold count must always be non-zero are marked with *HC>0.*

- States for which the hold count must always be non-zero and for which the hold count may never increase are marked with *HC--*.

Valid values for the extended level are represented as follows:

- States for which the extended level must always show a set of no levels are marked *XL=0.*

- States for which the extended level must show a set of levels consisting only of a single level are marked *XL=1.*

- States which for which the extended level may show any non-null set of compatible levels are marked *XL>0.*

- States for which the the extended level may show either a set of levels consisting of a single level or a null set of levels are makred *XL<=1.*

Valid values for the obtain mask are represented as follows:

- States for which the obtain mask must always be zero are marked *OM=0.*

- States for which the obtain mask may have any set of levels are marked *OM>=0.*

- States for which the obtain mask may not contain any levels compatible with the base level of the current extended level are marked *OM-I.*

Valid values for the revoke mask are represented as follows:

- States for which the revoke mask must always be zero are marked *RM=0.*

- States for which the revoke mask may have any set of levels are marked *RM>=0.*

TCCS_IDLE (*HC=0, XL=0, OM=0, RM>=0)*

The idle state is used when we do not have any token of the specified class and are not in the process of obtaining or returning one. Note that the revoke mask can be non-zero since we may have a revoke for an unsolicited token, i.e. one which we will note as possessed by calling tclient_gotten.

We may arrive at the idle state in the following ways:

- From the returning state (TCCS_RETURNING) when a tclient_returned call indicates that a return has been completed.

- From the cached state (TCCS_CACHED) in tclient_prepare_destroy.

- From the rejected state (TCCS_REJECTED) or the already state (TCCS_ALREADY) when an obtain fails either because of refusal or because we found on the server side that the token had already been obtained. These transitions will only occur when the hold count reaches zero indicating that all threads waiting for the obtain have taken account of the obtain failure.

From the idle state we can make the following state transitions:

- To the obtaining state (TCCS_OBTAINING) in tclient_obtaining. When making this transition, the hold count must be advanced from zero to one.

- To the cached state in tclient_gotten.

- To the returning state in tclient_gotten. This transition will occur if a revoke is pending for the token which is being marked as gotten.

TCCS_OWNED (*HC>0, XL>0, OM-I, RM*=0)

The owned state is used when we hold a token and it is not being revoked and there is no reason to return it even when its hold count reaches zero.

We may arrive at the owned state in the following ways:

- From the cached state (TCCS_CACHED) in tclient_obtaining. When making this transition the hold count must be advanced from zero to one.

- From the obtaining state in tclient_obtained.

From the owned state we can make the following state transitions:

- To the revoking state (TCCS_REVOKING) in tclient_revoke. This makes sure that the token will be returned when the hold count reaches zero.

- To the will-return state (TCCS_WILLRETURN) is tclient_returning. This makes sure that the thread waiting to do the return will be awoken to do the return when the hold count reaches zero.

- To the cached state in tclient_release if the hold count is going to zero.

- To the will-obtain state (TCCS_WILLOBTAIN) in tclient_returning when the hold count goes to zero and there is a pending bit in the obtain mask.

TCCS_CACHED (*HC=0, XL=1, OM=0, RM=0)*

The cached state is used when we posses but do not hold a token but are caching it for later reuse.

We may arrive at the cached state in the following ways:

- From the owned state in tclient_release if the hold count is going to zero.

- From the idle state in tclient_gotten.

- From the restore state when a rejection is cleared.

From the cached state we can make the following state transitions:

- To the owned state in tclient_obtaining.

- To the obtaining state in tclient_obtaining. This will occur when an attempt is made to obtain a token of another incompatible level.

- To the returning state in tclient_returning or tclient_revoke.

- To the idle state in tclient_prepare_destroy.

TCCS_BORROWED (*HC>0, XL>0, OM-I, RM=0)*

The borrowed state indicates that we hold a token, further acquires for the token will be granted, but that the token will be returned when the hold count goes to zero.

We may arrive at the borrowed state from the borrowing state (TCCS_BORROWING) in tclient_obtained.

From the borrowed state, we may make the following state transitions:

- To the revoking state (TCCS_REVOKING) in tclient_revoke. This makes sure that the token will be returned when the hold count reaches zero.
- To the will-return state (TCCS_WILLRETURN) is tclient_returning. This makes sure that the thread waiting to do the return will be awoken to do the return when the hold count reaches zero.
- To the returning state in tclient_release if the hold count is going to zero.
- To the will-obtain state in tclient_returning when the hold count goes to zero and there is a pending bit in the obtain mask.

TCCS_REVOKING (*HC--, XL>0, OM>=0, RM=0)*

The revoking state indicates that we hold a token which has been revoked. No further acquires will be granted and the token will be returned when the hold count goes to zero.

We may arrive at the revoking state in the following ways:

- From the owned state or the borrowed state in tclient_revoke.
- From the obtaining state in tclient_obtained if there is a revoke pending for the token level which is being marked as obtained.

From the revoking state we may make the following state transitions:

- To the returning state in tclient_release if the hold count is going to zero and the obtain mask is empty.
- To the will-obtain state in tclient_release if the hold count is going to zero and the obtain mask is non-empty.
- To the will-return state in tclient_returning.

TCCS_WILLRETURN (*HC--, XL>0, OM>=0, RM*=0)

The will-return state indicates that we hold a token which some thread is waiting to return. No further acquires will be granted and the waiting thread will be awoken when the hold count goes to zero.

We may arrive at the will-return state from the owned, borrowed, or revoking states in tclient_returning.

From the will-return state, the only valid transition is to the returning state in tclient_release when the hold count goes to zero.

TCCS_OBTAINING (*HC>0, XL>0, OM>=0, RM*>=0)

The obtaining state indicates that we are in the process of getting a token. One thread, the one which sets the state to obtaining is actually getting the token (for the base level associated with the current extended level) and will update the state upon completion of the obtain operation by calling tclient_obtained. Other threads establish their own holds updating the extended level just as would be done in the case of an acquire of a token we already posses. Such threads will wait in tclient_obtaining for the obtain to be completed.

We may arrive at the obtaining state in the following ways:

- From the idle state in tclient_obtaining.
- From the cached state in tclient_obtaining when we attempt to obtain a token for the same class with an incompatible level. In this case, we will return the token we have while obtaining the token are trying to get.
- From the will-obtain state in tclient_obtaining when we have to wait for our chance to obtain a token of the specified level. This will occur when tclient_obtaining encounters a situation which does not allow us to proceed immediately but in which we may set our bit in the obtain mask.

From the obtaining state we may make the following state transitions:

- To the owned state in tclient_obtained.
- To the borrowing state (TCCS_BORROWING) in tclient_borrowing.
- To the revoking state in tclient_obtained if a pending revoke is found for the token being marked as obtained.
- To the rejected or already state in tclient_obtained when a token obtain fails.
- To the restore state in tclient_acquire_conditional when a return of a token is rejected by the tcif_obtain routine.

TCCS_BORROWING (*HC>0, XL>0, OM-I, RM*>=0)

The borrowing state is similar to the obtaining state. It differs only in that once the token is obtained it will be returned whenever the hold counts goes to zero.

We may arrive at the borrowing state only from the obtaining state in tclient_borrowing.

From the borrowing state we may make the following state transitions:

- To the borrowed state in tclient_obtained.
- To the revoking state in tclient_obtained if a pending revoke is found for the token being marked as obtained.
- To the rejected or already state in tclient_obtained when a token obtain fails.
- To the restore state in tclient_acquire_conditional when a return of a token is rejected by the tcif_obtain routine.

TCCS_REJECTED (*HC>0, XL>0, OM-I, RM>=0*)

The rejected state indicates that an attempt to obtain a token was refused. Once all threads which have a hold on the token to be obtained have given that up, the token may advance to another state.

We may arrive at the rejected state only from the obtaining or borrowing states in tclient-obtained.

From the rejected state we may make the following state transitions:

- To the idle state when a rejection indication is cleared (this happens in tclient_obtained or the thread obtaining the token and tclient_obtaining for those waiting) and the hold count goes to zero and there is no thread waiting to obtain a token.

- To the will-obtain state when a rejection is cleared, the hold count reaches zero and there is a thread waiting to obtain a token.

TCCS_ALREADY (*HC>0, XL>0, OM-I, RM>=0*)

The already state indicates that an attempt to obtain a token failed due to the fact that the token server had already assigned us the specified token. We assume that a tclient_gotten is pending which will resolve the situation.

We may arrive at the already state only from the obtaining or borrowing states in tclient-obtained.

From the already state we may make the following state transitions:

- To the idle state when a rejection indication is cleared (this happens in tclient_obtained or the thread obtaining the token and tclient_obtaining for those waiting) and the hold count goes to zero and there is no thread waiting to obtain a token.

- To the will-obtain state when a rejection is cleared, the hold count reaches zero and there is thread waiting to obtain a token.

TCCS_RETURNING (*HC=0, XL=1, OM>=0, RM>=0*)

The returning state indicates that we are actually in the process of returning a token. The token may not be held.

We may arrive at the returning state in the following ways:

- From the idle state in tclient_gotten when the token being marked as gotten has already been rejected.

- In tclient_release from the revoking, borrowing, or will-return states when the hold count goes to zero.

- From the cached state in tclient_returning.

- From the restore state when a rejection of an obtain due to refusal (by the tcif_obtain routine) of a token return is being processed and the hold count goes to zero.

From the returning state we may make the following state transitions:

- To the idle state in tclient_returned.

- To the cached state in tclient_returned when revocation is refused by the tcif_revoke routine.

- To the will-obtain state if there is a pending bit set in the obtain mask.

- Back to the returning state when revocation is refused in the tcif_revoke routine and there is a pending revoke for the token being marked as refused.

TCCS_WILLOBTAIN (*HC=0, XL<=1, OM>=0, RM>=0)*

The will-obtain state indicates that a thread which had been waiting to obtain a specified token may now proceed. It is set by a thread which resets the obtain mask bit for the level which is now to be obtained. At this point the thread which originally set the corresponding obtain mask bit has leave to proceed to obtain the token.

The extended level may indicate a single level if there is a token to be returned with the one which is obtained. Otherwise, it indicates the null set.

We may arrive at the will-obtain state in the follow ways:

- From the owned, borrowed, and revoking states in tclient_release when the hold count is going to zero and there is a bit set in the obtain mask.

- From the returning state in tclient_returned when there is a bit set in the obtain mask.

- From the already and rejected states when a rejection is being cleared and the hold count goes to zero.

From the will-obtain state the only valid transition is to the obtaining state in tclient_obtaining.

TCCS_RESTORE (*HC>0, XL=1, OM>=0, RM>=0)*

The restore state is used to indicate that a token which was to be returned cannot be returned because the return was rejected by the tcif_obtain routine. Note that the extended level for this state is for the token whose returned failed, not the one to be obtained.

We may validly arrive at the restore state only from the obtaining or borrowing state in tclient_acquire_conditional.

From the restore state we may make the following state transitions:

- To the cached state when a rejection is cleared and the hold count goes to zero.

- To the returning state when a rejection is cleared, the hold count goes to zero and there is a pending revoke for the token in question.

11 Token Server Internals

The description of token server internals given here is fairly sketchy. A more complete description will be provided for the final version of the token server.

11.1 The Tsinfo Structure

Each server-side token controlled object is represented by a tsinfo structure. The tsinfo structure contains the following data items:

void * tsi_object

This gives the address of the object represented by this tsinfo. This address is presented to the user as one of the parameters to the tsif_ callouts. It is also available for debugging. It is not used otherwise since it is opaque to the token module.

tsintfc_t * tsi_intfc

This gives the address of the tsintfc corresponding to the current token-controlled object. This is used to determine the appropriate callout routine. The tsintfc can be used whenever there is information that should be stored for each object type rather than for each object.

char tsi_rstate

This gives the reclamation state for this tsinfo. The reclamation state consists of a status value to help us sequence through the revocation of active tokens necessary for reclamation. In addition it contains flag bits to determine whether we are actually in tserver_reclaim and whether the reclamation has failed.

char tsi_rclass

Indicates for which class reclamation is performing a revocation.

char tsi_lcount

Gives a count of token classes that are locked (See 11.2)

tscword_t tsi_cword[TCLASS_COUNT]

Provides a state word for each token class. The structure of the state word is given below in section 11.2.

int tsi_refs

Used for reference counting when the fact that the tsinfo is being referenced cannot be represented in the state word as a lock or a token possession (See 11.2).

udecl_simple_lock_data(,tsi_lock)

This lock governs all the variable data in the tsinfo structure.

tdebug_t * tsi_debug

This word, present only if the TSERVER_DEBUG option is on, points to a debugging structure which is used to trace activity on a given tsinfo.

11.2 The Token Server Class Word (tscword_t)

The class word contains a number of fields describing the status of tokens for a given token class. Each of the fields is defined by a mask of the bits within the fields. Macro's are defined to get and set each of these fields. The class word should never be referred to directly.

#define TSCW_MASK_LOCK 0x00000001

Indicates that the class is locked. The class must be locked to grant tokens or do a revoke.

#define TSCW_MASK_REVOKE 0x00000002

Indicates that a revocation is being done.

#define TSCW_MASK_TLEVEL 0x0000000c

Indicates the current token level. If there are no tokens currently assigned this has the value TLEVEL_INVALID.

#define TSCW_MASK_RECLAIM 0x00000010

Indicates that a revocation due to a reclaim is being done for this token.

#define TSCW_MASK_RCLPEND 0x00000020

Indicates that a revocation needs to be done on this token class to satisfy a reclaim request.

#define TSCW_MASK_NOTIFY 0x00000040

Indicates that the tsif_idle routine is to be called when there are no tokens held for this class.

#define TSCW_MASK_GRANT 0xff000000

Gives a mask of all of the token clients that have been granted a token of the specified class. The mask is accessed by a shift which is a client number which is converted from a tclient_t by TCN_GET_CNUM.

#define TSCW_MASK_REFUSE 0x00ff0000

Gives a mask of all of the token clients that have been refused a current token revocation for the specified class. The mask is accessed by s shift which is a client number which is converted from a tclient_t by TCN_GET_CNUM.

12 Guide to the Code

All of the code and headers for the token module are contained in the osf1_server/ad directory.

The following headers provide the exported interface that may be used by applications.

- **ad/token.h** defines all of the exported interface common to server and client. Types are defined for token class and for token level. Defines are present to construct token masks and also masks of token classes and token levels of a given class. Utility routines for handling token sets and associated data are provided. Finally, typedef's needed to compile the other headers are provided.
- **ad/tclient.h** defines the tcinfo and the tcintfc structures. It has prototypes for all of the exported routines of the token client module. This includes those necessary to provide token client functionality on a server acting as the token server. These latter routines are only present if the TOKEN_SERVER option is set.
- **ad/tserver.h** defines the tsinfo and tsintfc structures. It has prototypes for all of the exported routines of the token server module.

The following headers provide internal structure which should only be visible within the token module.

- **ad/token_imp.h** provides defines used by token client and server modules to scan token sets passed to it.
- **ad/tcl_imp.h** provides the items needed for the token client implementation. These include the definition of the structure of the token class word which summarizes the state of each token class, defines to manipulate the token class word, associated debugging defines, declarations of external symbols and the prototypes for subroutines internal to the token client implementation.

- **ad/tsrv_imp.h** provides the items needed for the token server implementation. Note that this is currently a very limited implementation which supports only two nodes.

- **ad/tdebug_imp.h** defines data structures used in a debugging package common to token client and token server. This provides services for recording the history of token operations in order to help find bugs which only become apparent long after the initial error.

The following source files implement the token client and token server modules:

- **ad/token.c** implements routines useful to users of both the token client and token server modules. This includes routines to deal with token sets and token data arrays.

- **ad/tdebug.c** implements routines used to provide debugging by both the token client and token server modules.

- **ad/tclient.c** implements all of the user-visible routines in the token-client interface and most of the internal routines used to implement these functions.

- **ad/tcl_debug.c** implements debugging support routines for the token client implementation.

- **ad/tcl_local.c** implements routines necessary to allow a local token-client-type interface on a server on which the token server for a given object is located. This includes the internal special callout routines in addition to the user-visible routines to set up this environment.

- **ad/tserver.c** implements all of the user-visible routines in the token-server interface and most of the internal routines used to implement these functions. Note that the current implementation is a temporary one limited to supporting two servers.

- **ad/tsrv_debug.c** implements debugging support routines for the token server implementation.

13 Acknowledgments

Paul Roy and Durriya Netterwala helped in forming the basic design ideas for the token module and as the first users had many helpful suggestions which led to an improved design and implementation.

References

[1] Roy, P., Noveck, D., Netterwala, D. The File System Architecture of OSF/1 AD Version 2. *OSF Research Institute Operating Systems Collected Papers*, Volume 3.