Open Software Foundation **R. Agarwalla (Transarc)**
Request For Comments: 73.0
October 1995

DFS TOKEN MANAGER REDESIGN

1. INTRODUCTION

A token encodes rights that allow the possessor to perform some operations on a particular object. Before performing any file operation, a DFS client obtains requisite tokens, from the file server, that grants the client the rights to perform that operation on the file. There are various types of tokens, some of which grant exclusive rights to the possessor and some shared rights. For exclusive tokens, such as the token that grants the holder permission to modify a specified range of the file, the holder can be certain that no other entity possesses the right to modify the file in the range specified by the token as long as the holder possesses the token. Before another entity will be granted the right to modify the same range of the file, a revocation of the token will be issued to the current holder. Tokens allow DFS to provide single site UNIX file semantics.

Tokens are managed on each file server by a token manager. The token manager is responsible for tracking tokens granted for each file on the server to different clients; ensuring no conflict between these granted tokens e.g., ensuring that two clients do not have tokens to modify the same range of a file simultaneously. It is responsible for issuing token revocation requests to the existing holder when another entity requests a token that conflicts with the currently granted token. It also has the responsibility of ensuring that the the conflicting token request is not granted until the current holder grants the revocation request.[1] It is responsible for queuing a token request that currently cannot be satisfied due to an outstanding token, if the client so requests, and informing that client when the token becomes available.

DFS also uses tokens to track changes to filesets.[2] Hence the token manager also manages the fileset tokens. The token manager is also responsible for ensuring consistency between file tokens and fileset tokens.

2. REDESIGN MOTIVATION

The existing token manager (TKM) is a good prototype that has served us well. But it can no longer satisfy the growing needs. A few of the problems with the existing TKM are described below:

(a) Complexity. The existing TKM has machinery for features that have been obsoleted by the changes and simplifications in the token usage and token recovery in event of client or server crashes or network outrages. This adds unnecessary overhead and performance penalties.

(b) Locking. The structure representing a file in the token manager has a very complex state machine associated with it. Some of the states represent a locked file while others an unlocked file. The transitions between these states are not done symmetrically. Some functions that caused a file to transition into the locked state do not put it in an unlocked state after completion of the intended operation. This leads to very high complexity of the code and an inability to verify that transitions into locked states are followed by transitions out of the locked states.

(c) Internal hash tables. The existing token manager has three hash tables to locate tokens. Of these, two hash tables organized by host and by token id are no longer required because of the improvements to token management in 1.0.3. The only hash table that is required is organized by file id. Internal code organization and locking would be simplified significantly by removing the obsolete hash tables.

---

1. On client crashes, network communication failure, it is possible that the token manager will revoke a token which the client will not know about till later. This falls under token state recovery which is outside the scope of this document.

2. Filesets are a logical collection of files residing together on logical unit of physical media.

(d) Maintainability of the existing TKM is becoming increasingly difficult because of the problems described and due to the numerous incremental enhancements to the module for new functionality to meet growing needs.

3. NEW DESIGN

The new design addresses the problems with the old TKM as described above. It is based on a well-defined small state machine. It has a well-defined locking hierarchy. There is a token garbage collection mechanism. Both file tokens and fileset tokens are handled similarly. The new token manager design allows for revocations of conflicting tokens from different clients in parallel as one of its core features. The following sections provide details about the new design and implementation.

4. DATA STRUCTURES

A token inside the TKM is represented as:

```
typedef struct tkm_internalToken {
    struct tkm_internalToken    *next;
    struct tkm_internalToken    *prev;
    hyper                       id;     /* token identifier */
    unsigned long               expiration; /* token expiration time
                                               */
    tkm_byterange_t             range;  /* for byte range token,
                                           represents range covered
                                           by token */
    tkm_tokenHolder_t           holder; /* pointer to the file or
                                           fileset id for which this
                                           token has been granted */
    long                        types;  /* token types granted
                                           e.g. DATA_READ,
                                           DATA_WRITE, STATUS_READ
                                           ... */
    long                        refused;/* token types revoked and
                                           refused */
    struct hs_host              *host;  /* client that requested this
                                           token */
    unsigned int                flags;  /* describes token state */
    unsigned long               refcount;
    unsigned long               reqNum;
    time_t                      lastRefused; /* time of last revoke
                                                refusal */
    struct tkm_internalToken    *nextGC; /* pointers to maintain GC
                                            list */
    struct tkm_internalToken    *prevGC;
} tkm_internalToken_t;

typedef struct tkm_tokenList {
    tkm_tokenMask_t             *mask; /* mask of all types of all
                                           tokens contained in this
                                           list */
    tkm_internalToken_p         list;  /* first token in this list */
} tkm_tokenList_t;
```

The above data structures reflect new optimizations in the new token manager for performance.

(a) If only one host has been granted tokens on an object, then the `host` field in the `tkm_internalToken_t` structure identifies that host. For a new token request from the same host, it allows us to bypass the potentially expensive token conflict resolution mechanism. The token conflict resolution mechanism is only called when the `host` field is nil which is if multiple hosts have been granted tokens on the object.

(b) The token conflict resolution mechanism has an additional optimization that allows for quick conflict detection. The `mask` field in the list of tokens `tkm_tokenList_t` contains the union of all token rights granted for the object. Hence if the requested token rights does not conflict with the mask of the granted token rights, the expensive token conflict resolution mechanism of checking against each granted token need not be done, a big savings.

A fileset is represented as:

```
typedef struct tkm_volume {
    struct tkm_volume   *next;
    struct tkm_volume   *prev;
    hyper               cell;
    hyper               id;              /* fileset id */
    int                 flags;           /* fileset type ReadOnly /
                                            ReadWrite */
    osi_dlock_t         lock;
    int                 tokenGrants;     /* since beginning of time */
    int                 refcount;        /* sum of files + tokens +
                                            active requests */
    tkm_tokenList_t     granted;         /* tokens granted (with mask)
                                          */
    tkm_internalToken_p beingGranted;    /* list of volume tokens in
                                            the process of being
                                            granted */
    tkm_internalToken_p queuedFileTokens; /* tokens for files
                                            belonging to this volume
                                            that were queued due to
                                            conflicts with the granted
                                            tokens of this volume */
    tkm_tokenMask_t     fileTokens;      /* Representing the sum of
                                            all types of granted
                                            tokens for files in this
                                            volume.  Given our locking
                                            hierarchy this is cheap to
                                            keep up to date */
    struct tkm_file     *files;          /* list of files in this with
                                            tokens */
    osi_dlock_t         fileLock;        /* lock protecting the file
                                            chain */
    struct tkm_volume   *gcNext;         /* thread for recycling
                                            candidates */
} tkm_vol_t;
```

A file is represented as:

```
typedef struct tkm_file {
    struct tkm_file     *hashNext;       /* global file hash table */
    struct tkm_file     *hashPrev;
    afsFid              id;              /* file id */
    osi_dlock_t         lock;
    int                 tokenGrants;     /* since beginning of time */
    int                 refcount;        /* sum of files + tokens +
                                            active requests */
```

```
    tkm_tokenList_t     granted;        /* tokens granted (with mask)
                                         */
    tkm_internalToken_p queued;         /* tokens queued */
    struct hs_host      *host;          /* NULL or only host that has
                                           been granted tokens for
                                           this file */
    tkm_vol_t           *vol;           /* volume containing this
                                           file */
    struct tkm_file     *next;          /* file list in containing
                                           volume */
    struct tkm_file     *prev;
    struct tkm_file     *gcNext;        /* thread for recycling
                                           candidates */
} tkm_file_t;
```

5. GLOBAL DATA

The new TKM maintains global data. Each global entity is protected by its lock. The various entities are:

(a) `tkm_volumeList` is list of volumes for which file tokens or volume tokens have been issued.

(b) `tkm_fileList` is a list of files for which tokens have been issued.

(c) `tkm_asyncTryQ` is a list of pending token requests that could not be granted immediately which should be satisfied when possible. This is called the asynchronous token grant queue.

(d) `tkm_freeTokens` is a list of unused tokens ready for use/reuse.

(e) `tkm_gcList` is a list of all granted tokens. When a token is granted, it is put at the end of this queue. A candidate token for recycling is taken from the queue head.

(f) `tkm_tokenCounter` tracks the number of allocated tokens in the system.

6. LOCKING

(a) `tkm_internalToken_t` — All fields in the token structure are protected by the holder's lock (or the tkm_freeTokens list's lock if the token is in the `tkm_freeTokens` list) except for `nextGC` and `prevGC` which are protected by lock for `tkm_gcList`.

(b) `tkm_vol_t` — All fields are protected by `lock` except for the following. The `files` is protected by the `tkm_fileLock` field in the structure. The `next`, `prev`, `refCount`, `gcNext` fields are protected by the global `tkm_volumelistLock`.

(c) `tkm_file_t` — All fields are protected by lock except the following. The `next`, `prev` are protected by the `fileLock` in the `tkm_vol_t` structure for the volume this file is contained in. The `refcount`, `hashNext`, `hashPrev` and `gcNext` by the global `tkm_fileListLock`.

6.1. Locking hierarchy

The order in which the various locks should be obtained is described below. If a thread needs to grab two locks, the lock earlier in the list needs to be obtained first. Apply this rule recursively for multiple locks.

(a) `tkm_file_t.lock`

(b) `tkm_vol_t.lock`

(c) `tkm_gcListLock`

(d) `tkm_freeTokenListLock`

(e) `tkm_tokenCounterLock.`

(f) `tkm_expirationLock` (This lock is needed to modify default token expiration time.)

(g) tkm_asyncTryQLock

(h) tkm_fileListLock

(i) tkm_volumeListLock

(j) tkm_vol_t.fileLock

(k) conflict.numElementsLock (This lock is used to protect data associated with parallel token revocation management.)

Also the lock protecting the host structure in the lower level host module falls last in the above locking hierarchy.

7. TOKEN STATES

A token can be in one of the following 4 states.

(a) FREE — Represents a file token or volume token is in the free token list.

(b) QUEUED_ON_FID — Represents a file token that could not be granted due to an outstanding granted conflicting file token and is queued to be granted when the conflicting token is returned or revoked. Such a token is in the tkm_file_t.queued list. When the token manager is able to grant the request for such a queued token, the grant is called an *asynchronous grant*.

(c) QUEUED_ON_VOL — Represents a file token that could not be granted due to an outstanding granted conflicting volume token and is queued to be granted when the conflicting volume token is returned. Such a token is in the tkm_vol_t.queuedFileTokens list. Such queued file tokens are also asynchronously granted by the token manager when possible.

(d) GRANTED — Represents a file token or a volume token that has been granted by the token manager. Such a token is also threaded into the token manager's token garbage collection list, tkm_gcList, for recycling attempts. A GRANTED token is threaded into the granted list of the owning tkm_file_t or tkm_vol_t object.

A token that has been asynchronously granted is also in the GRANTED state. Additionally the token is marked to be in the GRANTINGG substate while the asynchronous grant RPC is outstanding (not completed).

If a request for a byterange token conflicts with an already granted token, the token manager will attempt to revoke the conflicting token. The token manager will also attempt to offer up to two replacement byterange tokens to the client, to whom the revoke is issued, for the ranges within the currently granted range that do not conflict with the byterange of the new token request. Such replacement byterange tokens are called *slice-n-dice* tokens. While the RPC that grants the slice-n-dice tokens is outstanding, the slice-n-dice tokens will also be marked to be in the GRANTING substate.

The token manager also maintains two background threads; one recycles tokens for reuse and the other processes asynchronous grant requests respectively.

8. EXPORTED API

The token manager exported interface to the rest of DFS has three functions.

(a) tkm_GetToken() — The exported interface to request a token.

```
long tkm_GetToken(afsFid          *fileDescP,
                  long            flags,
                  afsToken        *tokenP,
                  struct hs_host  *hostP,
                  u_long          reqNumber,
                  afsRecordLock   *lockDescriptorP)
```

Input:

(i) `fileDescP` — the fid of file for which a token is being requested

(ii) `flags` — Optional values `QUEUE`, `NOCONFLICT`, `OPTIMISTIC`.

(iii) `hostp` — The host requesting the token.

(iv) `reqNumber` — The host's request number.

Output:

(i) `tokenP` — The token granted.

(ii) `lockDescriptorP` — Reason for a lock token grant failure.

Return Value:

(i) `TKM_SUCCESS`

(ii) `TKM_ERROR_REQUESTQUEUED` (not really a failure)

(iii) `TKM_ERROR_TOKENCONFLICT`

(iv) `TKM_ERROR_TOKENINVALID`

(v) `TKM_ERROR_INVALIDID`

(vi) `TKM_ERROR_FILEINVALID`

(vii) `TKM_ERROR_BADHOST`

(viii) `EROFS`

(ix) `TKM_ERROR_NOMEM`

(b) `tkm_ReturnToken()` — The exported interface to return all or a subset of the rights granted by a token.

```
long tkm_ReturnToken(afsFid     *fileDescP,
                     hyper      *tokenIDP,
                     hyper      *rightsP,
                     long       flags)
```

Input:

(i) `fileDescP` — The fid of file for which a token is being requested.

(ii) `tokenIDP` — The id of the token returned.

(iii) `rightsP` — The rights returned.

Return Value:

(i) `TKM_SUCCESS`

(ii) `TKM_ERROR_NOENTRY`

(iii) `TKM_ERROR_TOKENINVALID`

(c) `tkm_GetRightsHeld()` — The exported interface to query the rights granted to a particular client host for a specified file or volume. For a file, a byterange can be specified for read/write rights for data or locks.

```
long tkm_GetRightsHeld(struct hs_host *hostP,
                       afsFid *fileDescP,
                       hyper *startPosP,
                       hyper *endPosP,
                       hyper *rightsP)
```

Input:

(i) `fileDescP` — The fid of file for which a token is being requested.

(ii) `hostP` — The host that we are queried about.

(iii) `startPos` → `endPos` — The byte range that we are queried about.

Output:

(i) `rightsP` — The rights held.

Return Value:

(i) `TKM_SUCCESS`

(ii) `TKM_ERROR_NOENTRY`

9. IMPLEMENTATION

This section provides a pseudocode overview of the token manager module implementation.

9.1. External Interface APIs

```
tkm_GetToken(fileDescP, flags, tokenP, hostp, reqNumber,
             lockDescriptorP)
{
    /*
     * FreeTokens(fileDescP) is the set of token types that can
     * always be granted (for example READ_DATA for files in readonly
     * volumes) */
    if (type in FreeTokens(fileDescP))
        return SUCCESS;
    /*
     * maxTokens(fileDescP) is the set of token types that can
     * possibly be granted (for example WRITE_DATA is not in that set
     * for files in readonly volumes)
     */
    if (type not in maxTokens(fileDescP))
        return EROFS;

    if (IsVolDescriptor(fileDescP)) {
        vol = FindVol(fileDescP);
        if (vol == NULL)
            AddVol(fileDescP);
        CopyExternaltoInternal(tokenp, hostp, vol, internalToken);
        code = GetVolToken(internalToken, flags);
        ReleaseVol(vol);
    } else {
        file = FindFile(fileDescP);
        if (file == NULL)
            AddFile(fileDescP);
        CopyExternaltoInternal(tokenp, hostp, file, internalToken);
        code = GetFileToken(internalToken, flags);
        ReleFile(file);
    }

    if (code == SUCCESS)
        CopyInternaltoExternal(tokenp, internalToken)
    else
        if (code != ERROR_REQUESTQUEUED)
            RecycleToken(internalToken);
    return code;
```

```
}

tkm_ReturnToken(fileDescP, tokenIDP, rightsP, flags)
{
    if (type in FreeTokens(fileDescP))
        return SUCCESS;

    if (IsVolDescriptor(fileDescP)) {
        vol = FindVol(fileDescP);
        if (vol = NULL)
            return ERROR_NOENTRY;
        code = ReturnToken(vol, tokenId);
        ReleaseVol(vol);
    } else {
        file = FindFile(fileDescP);
        if (file = NULL)
            return ERROR_NOENTRY;
        vol = file->vol;
        code = ReturnToken(file, tokenId);
        ReleFile(file);
    }
    return code;
}

tkm_RightsHeld(hostP, fileDescP, startPos, endPos, rightsP)
{
    if (IsVolDescriptor(fileDescP)) {
        vol = FindVol(fileDescP);
        if (vol == NULL)
            return ERROR_NOENTRY;
        code = WhatRights(hostP, vol, startPos, endPos, rightsP);
        ReleaseVol(vol);
    } else {
        file = FindFile(fileDescP);
        if (file = NULL)
            return ERROR_NOENTRY;
        code = WhatRights(host, file, startPos, endPos, rightsP);
        ReleFile(file);
    }
    return(code);
}
```

9.2. Internal routines

```
/*
 * GetFileToken(newToken, flags) : Grants a file token
 *  Input:
 *     newToken = the token requested
 *     flags = nothing, QUEUED, or NOCONFLICT or OPTIMISTIC or
 *             ASYNCGRANT
 */
GetFileToken(newToken, flags)
{
    file = File(newToken);
    Lock(file);
```

```
/* Attempt to revoke any conflicting file tokens */
code = FixFileConflicts(newToken, file, flags, otherHostMask);
if (FAILED(code)) {
    /* In event of token conflict, queue request for async grant
     */
    if ((code == TOKEN_CONFLICT) && (flags & QUEUED)) {
        HoldFile(file);
        AddTokenToList(Queued(File(newToken)), newToken);
        Unlock(file);
        return REQUESTQUEUED;
    } else {
        Unlock(file);
        return FAIL;
    }
}

Lock(Vol(newToken));
wait until no conflict with volume tokens in process of being
granted;
/* Attempt to resolve conflicts with volume tokens */
code = tkm_FixVolConflicts(newTokenP, flags);
if (FAILED(code)) {
    /* In event of token conflict, queue request for async grant
     */
    if ((code == TOKEN_CONFLICT) && (flags & QUEUED)) {
        HoldFile(file);
        AddTokenToList(QueuedFileToken(Vol(newToken)), newToken);
        Unlock(Vol(newToken));
        Unlock(file);
        return REQUESTQUEUED;
    } else {
        Unlock(Vol(newToken));
        Unlock(file);
        return FAIL;
    }
}

if (new tokens granted for this file since beginning of this
    request)
    start over;

/* No unresolved conflicts - grant token */
GrantFileToken(newToken);

if (flags & ASYNCGRANT) {
    token.flags |= GRANTING;
    Unlock(vol);
    Unlock(file);
    Issue token grant RPC;
    Lock(file);
    Lock(vol);
    token.flag &= ~GRANTING;
    wakeup waiters sleeping on file;
}
```

```
    if (flags & OPTIMISTIC)
        add maximum possible additional rights not conflicting with
            outstanding tokens;
    Unlock(Vol(newToken));
    Unlock(file);
    return SUCCESS;
}

/*
 * GetVolToken(newToken, flags) : Grants a volume token
 *
 * Input:
 *    newToken = the token requested
 *    flags = nothing, QUEUED, or NOCONFLICT or OPTIMISTIC or
 *            ASYNCGRANT
 */
GetVolToken(newToken, flags)
{
    vol = Vol(newToken);
    Lock(vol);
    add token to list of volume tokens in process of being granted;
    /* Attempt to resolve any conflicting volume tokens */
    code = FixVolConflicts(newToken, flags);
    if (FAILED(code)) {
        remove token from list of volume tokens in process of being
            granted;
        Unlock(vol);
        wakeup anyone waiting for in progress volume tokens requests
            to complete;
        return FAIL;
    }
    if (no conflict with file tokens granted for file in the volume)
    {
        remove token from list of volume tokens in process of being
            granted;
        GrantVolToken(newToken);
        wakeup anyone waiting for in progress volume tokens requests
            to complete;
        return SUCCESS;
    }

    Unlock(vol);
    for (each file in the volume for which tokens have been granted)
    {
        Lock(file);
        code  = FixFileConflicts(newToken, file, flags);
        if (FAILED(code)) {
            Lock(vol);
            remove token from list of vol tokens in process of being
                granted;
            wakeup anyone waiting for in progress volume tokens
                requests to complete;
            Unlock(file);
            Unlock(vol);
            return FAIL;
```

```
        }
        Unlock(file);
    }
    Lock(vol);
    /*
     * While the above iteration was done, locks were dropped which
     * could result in new file tokens begin granted.  Hence check
     * for conflicts with file tokens again
     */
    if (conflict with new file tokens granted for file in the volume)
        start over;

    remove token from list of volume tokens in process of being
        granted;
    GrantVolToken(newToken);
    wakeup anyone waiting for in progress volume tokens requests to
        complete;
    Unlock(vol);
    return SUCCESS;
}

/*
 * tokenList_t AllocToken(n): returns a list of n free tokens
 */

AllocToken(n)
{
    allocedTokensList = NULL;
    Lock(freeTokenList);
    while (numOfTokens(allocedTokensList) < n) {
        if (you find a token in freeTokenList)
            move token from freeTokenList to allocedTokensList;
        else {
            Lock(tokenCounter);
            /* Check if we have hit the token number limit */
            if (tokenCounter < tokenQuota) {
                create tokens;
                adjust tokenCounter;
                add the created tokens to the freeTokenList;
                Unlock(tokenCounter);
            } else {
                Unlock(tokenCounter);
                if (token garbage collection is unable to reclaim
                    tokens) {
                    Unlock(freeTokenList);
                    return NULL;
                } else {
                    trigger the token garbage collection (recycle)
                        thread;
                    wait for tokens to become free;
                }
            }
        }
    }
```

```
    Unlock(freeTokenList);
    return(allocedTokensList);
}

/*
 * TokenGcThread(): This is the single thread that does GC.  When
 * woken up it will keep running until there is at least
 * MIN_TOKENS_FREE tokens in the freeTokenList.
 */

TokenGcThread()
{
    while (TRUE) {
        Lock(freeTokenList);
        while (numOfTokens(freeTokenList) < MIN_TOKENS_FREE) {
            Unlock(freeTokenList);
            Lock(gcList);
            token = head(gcList);
            while ((token != NULL) && (revocation of token failed
                   recently))
                token = token->GCnext;
            if (token == NULL) {
                /*
                 * all of the tokens in the GClist have been already
                 * revoked and the revocations have been refused
                 */
                Lock(freeTokenList);
                set up indication that the token garbage collection
                    has is unable to reclaim tokens;
                Unlock(gcList);
                wakeup sleepers waiting for tokens to become free;
                break;
            }

            /* Gather candidate tokens to attempt to reclaim */
            while (numOfTokens(tokensToGc) < MAX_REVOKEINPARALLEL) {
                lastHeldToken = 0;
                tokenId = Id(token);
                if (IsFileToken(token)) {
                    HoldFile(File(token));
                    holder = File(token);
                } else {
                    HoldVol(Vol(token));
                    holder = Vol(token);
                }
                Unlock(gcList);
                Lock(holder);
                Lock(gcList);
                if (Id(token) == tokenId) {
                    HoldToken(token);
                    Unlock(holder);
                    AddToTokenList(tokensToGC, token);
                    lastHeldToken = token;
                } else {
                    /*the token has already been recycled*/
```

```
                    Unlock(holder);
                    if (lastHeldToken != NULL)
                        token = lastHeldToken->GCnext;
                    else
                        token = head(gcList);
                    while (revocation of token failed recently)
                        token = token->nextGC;
                }
            }
            Unlock(gcList);
            /*
             * RevokeInParallel will put tokens that could be
             * reclaimed onto the free token list.
             * At the end of the call, tokensToGC now contains only
             * those tokens which could not be revoked.  The tokens
             * that were revoked are added into the freeTokenList by
             * the call.
             */
            RevokeInParallel(tokensToGC, NULL);
            ReleTokenList(tokensToGC);
            Lock(freeTokenList);
        }
        SleepAndUnlock(freeTokenList, freeTokenListLock);
    }
}

/*
 * FixFileConflicts(token, file, flags, otherhostsmask) - attempt to
 * revoke granted file tokens conflicting with the requested token.
 *
 * INPUT:
 *      token - the token that we want to grant
 *      file - file for which the token is desired.
 *      flags - NOCONFLICT | QUEUED
 *
 * OUTPUT:
 *      otherhostsmask - mask of token types not conflicting
 *          with token but granted to hosts other than token->host
 *
 * LOCKING:
 *      Called with file locked.  Vol(token) MUST be unlocked.  The
 *      file lock might be dropped and reobtained
 *
 * GENERAL:
 *      If this routine returns SUCCESS it guarantees that until the
 *      file lock is dropped again there will be no granted tokens
 *      conflicting with the token passed in as the first input
 *      parameter.  If flags included QUEUED and a FAILURE is
 *      returned due to conflict, there is a guarantee that until the
 *      file lock is dropped there will be in the granted list of the
 *      file at least one conflicting token that has been sent a
 *      revocation (so it's safe to queue the token requested)
 */
FixFileConflicts(token, file, flags, otherhostsmask)
{
```

```
sliceDice = NULL;

/* If the only host to which tokens have been issued to for the
 * file is the same as the host requesting the token, there is no
 * conflict
 */
if (token->host == file->host)
    return SUCCESS;

while ((tokensToRevoke =
        Conflicting(granted(file), token, otherhostmask,
                    MAX_REVOKEINPARALLELS)) != NULL) {
    if (flags & (NOCONFLICT)) {
        ReleTokenList(tokensToRevoke);
        return FAILURE;
    }
    if (sliceDiceNeeded(tokensToRevoke, token) >
        numOfTokens(sliceDice)) {
        Unlock(file);
        sliceDice = AllocToken(sliceDiceNeeded(tokensToRevoke,
                    token) - numOfTokens(sliceDice));
        Lock(file);
        ReleTokenList(tokensToRevoke);
        continue;
    }

    if (sliceDiceNeeded(tokensToRevoke, token) > 0) {
        fill sliceDice list;
        return excess to freeTokenList;
        foreach (slice and dice token in sliceDice) {
            /* sltoken is the slice and dice token */
            sltoken.flag |= GRANTING;
            Lock(file->vol);
            GrantFileToken(sltoken);
            Unlock(file->vol);
        }
    } else
        if (sliceDice)
            put all tokens in sliceDice list to freeTokenList;
    Unlock(file);
    code = RevokeInParallel(tokensToRevoke, sliceDice);
    /* At this point sliceDice tokens contains the sliceDice
     * not accepted, and tokensToRevoke the tokens that were
     * refused */
    Lock(file);
    Lock(vol);
    put all sliceDice tokens to freeTokenList;
    wakeup waiters of sliceDiceTokens (sleeping on file);
    if (FAILED(code)) {
        if ((code == TOKENCONFLICT) && (flags & QUEUED))
            /* We dropped locks above.  Hence we need to recheck
             * if any conflict still exists. */
            if (any token in tokensToRevoke conflicts with
                newToken) {
                ReleTokenList(tokensToRevoke);
```

```
                    return TOKENCONFLICT; /* some tokens were refused
                                           */
                } else {
                    /* the refused tokens have been returned  so try
                       again */
                    ReleTokenList(tokensToRevoke);
                }
            else {
                ReleTokenList(tokensToRevoke);
                return FAILURE;  /* some tokens were refused */
            }
        }
    }
    /* we exited the while so there are no more conflicting tokens*/
    return SUCCESS;
}


/*
 * FixVolConflicts(token, flags, otherHostMask) - attempt to revoke
 *  granted volume tokens conflicting with the requested token.
 *
 * INPUT:
 *      token - the token that we want to grant
 *      flags - NOCONFLICT | QUEUED
 *
 * OUTPUT:
 *      otherHostMask - a mask of token types not conflicting with
 *        token but granted to hosts other than token->host
 *
 * LOCKING:
 *      lock of file for token (if not NULL) and lock of volume for
 *      token are held.  Note that file is NULL for volume tokens.
 *
 * GUARANTEES:
 *      If this routine returns SUCCESS it guarantees that until the
 *      vol lock is dropped again there will be no granted vol tokens
 *      conflicting with the token passed in as the first input
 *      parameter.  If flags included QUEUED and a FAILURE is
 *      returned, it guarantees that until the vol lock is dropped
 *      there will be in the granted list of the vol at least one
 *      conflicting token that has been sent a revocation (so it's
 *      safe to queue the token requested)
 */

FixVolConflicts(token, flags, otherHostMask)
{
    while ((tokensToRevoke =
            Conflicting(granted(vol), token, otherhostmask,
                        MAX_CONFLICT)) != NULL) {
        if (flags & (NOCONFLICT)) {
            ReleTokenList(tokensToRevoke);
            return FAILURE;
        }
        if (IsFileToken(token))
```

```
            Unlock(File(token));
        Unlock(Vol(token));
        RevokeInParallel(tokensToRevoke, NULL);
        if (IsFileToken(token)) {
            Lock(File(token));
            Lock(Vol(token));
            if (tokensToRevoke != NULL)
                if (flags & QUEUED)
                    if (any token in tokensToRevoke conflicts with
                        newToken) {
                        ReleTokenList(tokensToRevoke);
                        return FAILURE;
                    } else
                        ReleTokenList(tokensToRevoke);
                else {
                    ReleTokenList(tokensToRevoke);
                    return FAILURE;
                }
        } else {
            Lock(Vol(token));
            if (tokensToRevoke != NULL) {
                ReleTokenList(tokensToRevoke);
                return FAILURE;
            }
        }
    }
    /* we exited the while so there are no more conflicting tokens */
    return SUCCESS;
}

/*
 * Conflicting(tokenList, token, otherHostMask, maxnum) - finds
 * all the tokens in the tokenList that concflict with the requested
 * token.
 *
 * INPUT:
 *      tokenList - a token list representation
 *      token - the token that we want to grant
 *      maxnum - the maximum number of tokens to return.
 *
 * OUTPUT:
 *      otherHostMask - a mask of token types not conflicting with
 *      token but granted to hosts other than token->host.  This mask
 *      might include token types even if the only tokens granted for
 *      this type are granted to token->host The above will happen if
 *      this routine can determine that there are no conflicts
 *      without traversing the token list (i.e., by simply looking at
 *      the tokenDescription's mask)
 *
 * LOCKING:
 *      called with holder locked.
 */
tokenList_t Conficting(tokenDescription, token, otherHostMask,
                       maxnum)
{
```

```
    /*
     * First tries possiblyConflicting() which uses the token mask
     * and always returns 1 if a conflict exists and might return 0
     * or 1 if no conflicts exist.  The erroneous 1 will be returned
     * if a conflict exists only with a token granted to the same
     * host.  If possiblyConflicting returns 1 the routine determines
     * conflicts by traversing the list of tokens.
     */

    conflictingTokenList = NULL;
    if (!possiblyConflicting(mask(tokenDescription), token)) {
        otherHostMask = ~(mask(tokenDescription));
        return NULL;
    }
    foreach (token in tokenList) {
        /* tltoken represents token in tokenList */
        if (tltoken conflicts with token) {
            AddToTokenList(conflictingTokenList, tltoken);
            if (numOfTokens(conflictingTokenList) == maxnum)
                return conflictingTokenList;
            else
                if (tltoken->host != token->host)
                    otherHostMask |= tltoken->type;
        }
    }
    return conflictingTokenList;
}

/*
 * RevokeInParallel(tokensToRevoke, sliceNdice) - revoke tokens and
 * possibly simultaneously offer replacement slice-n-dice tokens.
 *
 * INPUTS:
 *      tokensToRevoke - On entry this has the tokens that need to be
 *        revoked.  All these tokens are held.  On exit the list
 *        contains the tokens that were refused (still held).  The
 *        tokens that were succesfully revoked are freed.  The caller
 *        must make sure that the list contains at most
 *        MAX_REVOKEINPARALLEL tokens.
 *
 *      sliceNdice - On entry it is null or has the list of tokens to
 *        be offered (could be NULL).  On exit the list contains the
 *        ones that were not accepted.  Again all tokens are held on
 *        entry and only the ones that are accepted are released here
 *        (because they are removed from the list.
 *
 * LOCKING:
 *      Must be called without any locks held since it will probably
 *      issue RPCs.
 *
 * GENERAL:
 *      tokens that have the GRANTING flag on cannot be revoked, so
 *      this routine has to sleep.
 */
```

```
RevokeInParallel(tokensToRevoke, sliceNdice)
{
    foreach (token in tokensToRevoke that is expired) {
        fileToken = IsFileToken(token);
        if (fileToken) {
            tokenFileHolder = File(token);
            Lock(tokenFileHolder);
        }
        tokenVolHolder = Vol(token);
        Lock(tokenVolHolder);
        remove token from tokensToRevoke list;
        ReleToken(token) ;
        if (fileToken)
            Unlock(tokenFileHolder);
        Unlock(tokenVolHolder);
    }

    sort the rest of the tokens by most efficient way to do
        revocations;
    issue RPCs for the tokens to be revoked in the tokensToRevoke
        list;
    /*
     * The above will sleep waiting for tokens that have the GRANTING
     * flag set.  Also it will not send RPCs for the tokens on which
     * revocations have been refused in the immediate past for
     * efficiency.
     */

    revokedTokens = tokens succesfully revoked;
    foreach (token in revokedTokens) {
        fileToken = IsFileToken(token);
        if (fileToken) {
            tokenFileHolder = File(token);
            Lock(tokenFileHolder);
        }
        tokenVolHolder = Vol(token);
        Lock(tokenVolHolder);
        remove token from tokensToRevoke list;
        FreeToken(token);
        if (fileToken)
            Unlock(tokenFileHolder);
        Unlock(tokenVolHolder);
    }

    foreach (slice-n-dice token offered) {
        /* sltoken represents the slice-n-dice token */
        if (sltoken accepted)
            sltoken.flag &= ~GRANTING; /* wakeup done by our caller
                                          */
        else {
            Lock(sltoken->holder);
            FreeToken(sltoken);
            Unlock(sltoken->holder);
        }
    }
```

```
}

/*
 * GrantFileToken(token) - grant a file token
 *
 * LOCKING:
 *      File(token) and Vol(token) are locked
 */
GrantFileToken(token)
{
    File(token)->tokenGrants++;
    Vol(token)->tokenGrants++;
    HoldFile(File(token));
    AddTokenToList(granted(File(token)), token);
    Lock(gcList);
    AddTokenTogcList(token);
    Unlock(gcList);
    AddToTokenMask(fileTokens(Vol(token)), types(token));
}

/*
 * GrantVolToken(token) - grant a volume token
 *
 * LOCKING:
 *      Vol(token) is locked
 */

GrantVolToken(token)
{
    vol->tokenGrants++;
    HoldVol(Vol(token));
    AddTokenToList(granted(Vol(token)), token);
    Lock(gcList);
    AddTokenTogcList(token);
    Unlock(gcList);
}

/*
 * TriggerAsyncGrants(tokenP) - This routine simply copies a part of
 * the token just revoked to an element of the tkm_asyncTryQ and
 * wakes up the thread that will read this element and decide whether
 * there are any tokens waiting to be granted that were blocked on
 * the token just revoked (tokenP).
 *
 * INPUT:
 *      tokenP - the token just revoked
 *
 * LOCKING:
 *      tokenP's holder must be locked
 */

TriggerAsyncGrants(tokenP)
{
    new = get asyncGrantTryQ element;
    copy relevant information including token object holder, host,
```

```
        type from token to the the new asyncGrantTryQ element;
    Lock(asyncGrantTryQ);
    enqueue new onto asyncGrantTryQ
    Unlock(asyncGrantTryQ);
    wakeup asyncGrantThread if it is sleeping;
}

/*
 * AsyncGrantThread() - This is the thread that takes care of async
 *  grants it is woken up by whoever puts a file or a vol in the
 *  TryasyncGrantTryQ queue.  Once woken up it will continue running
 *  until the queue is empty
 */
AsyncGrantThread()
{
    while (TRUE) {
        Lock(asyncGrantTryQ);
        foreach (entry on asyncGrantTryQ) {
            fileP = volP = NULL;
            revokedTypes = entry.types;
            tokenHost = entry.host;

            if (entry for file token)
                fileP = entry.file;
            else
                volP = entry.vol;

            Unlock(asyncGrantTryQ);

            if (fileP) {
                Lock(fileP);
                TryAsyncGrantsOnList(Queued(fileP), revokedTypes,
                    tokenHost);
                Unlock(fileP);
                ReleFile(file);
            } else {
                Lock(volP);
                TryAsyncGrantsOnList(QueuedFileToken(volP),
                                   revokedTypes, tokenHost);
                Unlock(volP);
                ReleVol(volP);
            }
            Lock(asyncGrantTryQ);
        }
        SleepAndUnlock(asyncGrantTryQ, asyncGrantTryQ.lock);
    }
}

/*
 * TryAsyncGrantsOnList(listP, host, revokedTypes, lockP) - This
 * routine traverses a list of tokens and sees if any can be granted
 * as a result of a recent token revocation.
 *
 * INPUT:
 *      listP -the list of candidates to be granted
```

```
 *      host - the host that held the token who's revocation
          triggered this async grant attempt
 *      revokedTypes - the rights that were revoked from the token
 *        who's revocation triggered this async grant attempt
 *      lockP = the lock protecting listP
 *
 * LOCKING:
 *      lockP must be held
 */
TryAsyncGrantsOnList(listP, host, revokedTypes, lockP)
{

    foreach (token on input list) {
        assert(token is a file token);
        if ((token->host != host) &&
            (token->type conflicts with revokedTypes)) {
                RmTokenFromList(listP, token);
                Unlock(lockP);
                /* At this point the token still holds a reference to
                 * its file and is not in any list so it can't change
                 * from under us.
                 * This will attempt to get the token.  If unable to,
                 * the request will be queued again
                 */
                GetFileToken(token, QUEUE | ASYNCGRANT);
                Lock(lockP);
        }
    }
}

/*
 * ReturnToken(holder, tokenId, rights) - return a token
 *
 * INPUTS
 *      holder - can be a vol or a file.  It's the token's
holder.  It's
 *        held and will be released by our caller.
 *      tokenId - the returned token id
 *      rights - the rights returned.
 *
 * LOCKING:
 *      nothing is locked
 */
ReturnToken(holder, tokenId, rights)
{
    Lock(holder);
    tokenList = granted(holder).tokens;
    foreach (token in tokenList) {
        if (Id(token) = tokenId) {
            HoldToken(token);
            FreeToken(token);
            break;
        }
    }
```

```
    /*
     * Even if no matching token found, return success.  Client may
     * be returning a token that we have already recycled.  This
     * could happen because of revocations due to network
     * communication problems, client crash or token expiration.
     */
    Unlock(holder);
    return SUCCESS;
}

/*
 * WhatRights(host, holder, startPos, endPos, rights) - determine the
 * token rights granted to the specified host on the specified
 * object.
 *
 * INPUT:
 *      host - the host who's rights we are queried about
 *      holder - can be a vol or a file.  It's the object for which
 *        we are trying to find out the rights that the host has.
 *        It's held and will be released by our caller.
 *      startPos, endPos: the boundaries of the byte range that we
 *        are queried about.  These are only meaningful if
 *        type(holder) == FILE
 *
 * OUTPUT:
 *      rights - the rights held
 *
 * LOCKING:
 *      nothing is locked
 */

WhatRights(host, holder, startPos, endPos, rights)
{
    Lock(holder);
    tokenList = Granted(holder).tokens;
    foreach (token in tokenList) {
        if (token->host == host) {
            rights |= (token.types & ~BYTERANGETOKENTYPES);
            if ((holder is a file object) &&
                (InRange(token.byteRange, startPos, endPos)))
                rights |= (token.types & BYTERANGETOKENTYPES);
        }
    }
}

/*
 * Utilities for token recycling.
 *
 * LOCKING:
 *      All these utility routines must be called with the holder's
 *      lock held.
 */

FreeToken(token)
{
```

```
    ReleaseToken(token, token->types);
}

ReleToken(token)
{
    ReleaseToken(token, 0);
}

HoldToken(token)
{
    token->refcount++;
}

ReleaseToken(token, types)
{
    if (token->types & types) {
        token->types &= ~types;
        if (IsFileToken(token))
            SubtractFromMask(fileTokens(Vol(token)), token);
        if (!(token->flags & GRANTING))
            TriggerAsyncGrants(tokenP);
    }

    token->refcount --;
    if ((token->refcount == 0) && (token->types == 0)) {
        if (IsFileToken(token)) {
            RmTokenFromList(granted(File(token)), token);
            SubtractFromMask(fileTokens(Vol(token)), token);
            Lock(gcList)   ;
            RmTokenFromGcList(token);
            Unlock(gcList);
            ReleFile(File(token));
        } else {
            RmTokenFromList(granted(Vol(token)), token);
            Lock(gcList)   ;
            RmTokenFromGcList(token);
            Unlock(gcList);
            ReleVol(Vol(token));
        }
        Lock(freeTokenList);
        since a token has been freed, reset any indication of token
            garbage collection failure to reclaim tokens;
        PutTokenInFreeList(token);
        wakeup sleepers of freeTokenList;
        Unlock(freeTokenList);
    }
}

/*
 * Utilities for file and vol
 */

HoldFile(file)
{
    Lock(fileList);
```

```
    file->refcount++;
    Unlock(fileList);
}

ReleFile(file)
{
    Lock(fileList);
    file->refcount++;
    Unlock(fileList);
}

HoldVol(vol)
{
    Lock(volumeList);
    vol->refcount++;
    Unlock(volumeList);
}

ReleVol(vol)
{
    Lock(volumeList);
    vol->refcount--;
    Unlock(volumeList);
}
```

10. TESTING

We have tested the new implementation in several ways. The token manager resides in the kernel on the file server machine.

10.1. User space

To facilitate testing, the token manager was enhanced to allow it to run in user space for testing and several user space tests developed.

(a) `tkm_test_units` tests the most crucial functionality of the token manager, the detection of token conflicts. It tests for conflict between two tokens as well as conflict between a requested token and a list of granted tokens. It tests byterange tokens too.

(b) `tkm_systest` has several modes to test the asynchronous grant mechanism, parallel requests for non-conflicting and conflicting tokens. This test for race conditions as well as internal data structure leakages. TKM exports three `pioctl()`'s to do `tkm_GetToken()`, `tkm_ReturnToken()` and `tkm_GetRightsHeld()`. This program will start numerous parallel threads performing these operations on different files and volumes. To test the results it will maintain internally a list of all the files and tokens that it has tokens for. Every time that a thread gets or returns a token the program will update its internal lists. After updating it will test to ensure that the state of the system is what it expects it to be. The update and test will be done entirely while holding a single global lock.

10.2. Kernel space tests

Several tests on a running DFS system were helpful in ensuring reliability of the new token manager running in its native mode in the kernel.

(a) Run cache consistency tests along with mmap based reader/writer tests to test handling of multiple parallel token requests that may conflict with each other.

(b) On a two machine cell, with machine 1 being the server for a fileset, create a directory with a large of number of different sized files. On machine 2, copy the entire directory to another directory in the same fileset, then delete the directory. Simultaneously on machine 1, run a program like `du` that recursively stats all the file in that that directory. This tests parallel requests for conflicting tokens, as well as system behaviour under load depending on the number of files used and the token allocation settings of the token manager.

(c) On a two machine, run the move-tsr tests whole volume token operations.

In addition to the successfull execution of the above functionality tests, the ability of the new token manager to withstand the extreme stress placed on it by the tests run by our system test group has shown that the stability of the new token manager greatly exceeds that of its predecessor. Also measurements have shown that the performance of the new token manager has greatly improved.

11. ACKNOWLEDGEMENTS

Thanks are due to C. Everhart (Transarc) for patiently answering my queries about tokens in general and to M. L. Kazar (now at FORE Systems) for fielding my questions even in his new life. D. Varotsis (now at FORE Systems) was instrumental in early drafts of this document.

AUTHOR'S ADDRESS

Rajesh Agarwalla
Transarc Corporation
The Gulf Tower
707 Grant Street
Pittsburgh, PA 15219
USA

Internet email: rajesh@transarc.com
Telephone: +1-412-338-4400