



CRESCO

**Centro computazionale di RicErca sui Sistemi
COMplessi**

**Dipartimento di Ingegneria dell'Informazione e Matematica
Applicata**

Università di Salerno

Responsabile Unità Operativa

Prof. Ciro D'Apice

LA1-SP I.2 - Sviluppo ed integrazione dell'architettura GRID e di grafica 3 D

Attività I.2.1 Integrazione ENEA GRID con altre architetture di GRID computing

1b – Studio e definizione del sottosistema di gestione utenti, gestore di Policy delle VO e loro integrazione con ENEA –GRID

Studio di fattibilità per il porting in ambiente Windows delle routine gettok
e puttok del kit LSF per AFS

V. 1.0

Indice

1. Varie	4
1.1. Bibliografia e Riferimenti	4
1.2. Acronimi.....	4
1.3. Glossario	5
2. Contesto, obiettivi e scopo	7
3. Le API per AFS	9
4. Platform LSF Source Code Package for AFS.....	11
4.1. Logica di impiego delle routine del package	11
4.1.1. Il metodo esub.....	11
4.1.2. Il metodo eexec	13
4.2. La routine gettok.....	14
4.2.1. La routine getAFSTokens	15
4.2.2. La routine printToken	18
4.3. La routine puttok	21
4.3.1. La routine converAFSTokens	21
4.3.2. La routine putAFSTokens.....	25
4.4. La routine rtok	26
4.5. Il server rtokd.....	26
4.6. Una visione d'insieme	26
5. OpenAFS per Windows	30
5.1. Strutture dati e function prototypes	30
5.2. La lettura del token.....	32
5.2.1. La routine ktc_ListTokens.....	32
5.2.2. La routine ktc_GetToken.....	35
5.3. La scrittura del token.....	39
5.3.1. La routine ktc_SetToken	39
Appendice A -Programming with the Andrew Cache Manager.....	43
1. Introduction	44
2. Semantic Differences.....	44

2.1.	Intro.....	44
2.2.	File Protection and Access Control Lists.....	44
2.3.	Open Modes.....	45
2.4.	File Consistency.....	46
2.5.	SetUID Programs.....	46
2.6.	Locking Files.....	46
2.7.	Hard Links.....	46
2.8.	Volumes, Partitions and Quotas.....	46
2.9.	Errors.....	47
3.	Special Functions.....	47
3.1.	IOCTLs.....	48
3.2.	PIOCTLs.....	48
3.3.	Complete list of AFS Assigned Numbers Registry - ioctl() and piocctl() function codes.....	55
Appendice B -Variabili di ambiente disponibili nell'ambiente di esecuzione dell'esub.....		60
Appendice C -Compilazione ed installazione di OpenAFS in ambiente Windows.....		63

Indice delle Figure

<u>Figura 1 - Strutture dati coinvolte nella routine getAFSTokens.....</u>	<u>17</u>
<u>Figura 2 - Strutture dati coinvolte nella routine printTokens.....</u>	<u>20</u>
<u>Figura 3 Strutture dati coinvolte nella routine converAFSTokens.....</u>	<u>24</u>
<u>Figura 4 Strutture dati coinvolte nella routine ktc_ListTokens.....</u>	<u>34</u>
<u>Figura 5 Strutture dati coinvolte nella routine ktc_GetTokens.....</u>	<u>38</u>
<u>Figura 6 Strutture dati coinvolte nella routine ktc_setToken.....</u>	<u>42</u>

1. Varie

1.1. Bibliografia e Riferimenti

- [1] Analisi dell'infrastruttura GRID ENEA ed individuazione dei punti critici ai fini dell'interoperabilità e dell'estensione delle caratteristiche di gestione e controllo dell'infrastruttura stessa
- [2] AFS User Guide di IBM
- [3] Open AFS Guide
- [4] Programming with the Andrew Cache Manager
- [5] Administering Platform LSF
- [6] <http://www.cresco.enea.it>
- [7] <http://www.platform.com>
- [8] <http://www.openafs.org>

1.2. Acronimi

ACL	A ccess C ontrol L ist
AFS	A ndrew F ile S ystem
API	A pplication P rogram I nterface
CRESCO	C entro C omputazionale di RicErca sui Sistemi Complessi
IOCTL	I nterface O utput C ontrol
KDC	K ey D istribution C enter
KTC	K ernel T oken C ache
KVNO	K ey V ersion N umber
LSF	L oad S haring F acility
PAG	P rocess A uthentication G roup
RPC	R emote P rocedure C all
TGS	T icket G ranting S ervice
TGT	T icket G ranting T icket
UID	U ser I dentifier

1.3. Glossario

Authentication Service (AS)

E' la parte di KDC che distribuisce i ticket sia i TGT sia altri tipi di ticket per l'accesso ai servizi.

Cache Manager

In AFS è il processo che gira sui sistemi client, trasferisce i dati da e verso il server, gestisce l'accesso, la visualizzazione, e la modifica dei file dalla directory principale /afs che rappresenta la radice di una struttura organizzata ad albero.

Clear Token

Parte del token di AFS che contiene al suo interno, fra l'altro, la Session Key (anche detta Handshake Key). Dai codici sorgenti esaminati la lunghezza del Clear Token dovrebbe essere di 24 byte.

Complex Mutual Authentication

Particolare tipo di Mutual Authentication che coinvolge tre chiavi di cifratura e tre parti. E' il tipo di autenticazione adottato da AFS

Handshake Key

Vedere Session Key.

Kerberos Authentication Server (kaserver)

E' uno dei processi server di AFS ed assicura che le comunicazioni sulla rete siano sicure. Verifica l'identità dell'utente al login e fornisce le strutture con le quali i partecipanti nelle transazioni verificano la loro identità uno rispetto all'altro. Gestisce l'Authentication DB.

KDC (Key Distribution Center)

E' il servizio che implementa l'autenticazione Kerberos mediante l'Authentication Service (AS) ed il Ticket Granting Service (TGS). Il KDC ha una copia di ogni encryption key associata a ciascun principal. La maggior parte delle implementazioni di KDC archivia i principals in un database e quindi a volte si impiega il termine Kerberos database applicato al KDC.

Mutual Authentication

Meccanismo con il quale le parti provano la loro effettiva identità e basato sul fatto che le parti provano la loro identità mostrando la conoscenza di uno shared secret.

Principal

E' il nome con cui si fa riferimento alle *entry* nel database dell'Authentication Server. Ad ogni utente, host o servizio di un dato realm viene associato un principal. Il generico formato

di un principal utente è : Nome[/Istanza]@REALM mentre per un servizio è Servizio/Hostname@REALM

Realm

(letteralmente regno) indica un dominio amministrativo di autenticazione che fissa i confini entro cui un server di autenticazione è autoritario nell'autenticare un utente, un host o un servizio.

Secret Key

Una chiave di cifratura permanente condivisa da un principal e dal KDC, impiegata per cifrare/decifrare la Session Key inclusa nel TGT dell'autenticazione iniziale. Nel caso il principal sia un utente la Secret Key è derivata dalla password Kerberos dell'utente stesso.

Secret Token

Parte del token riferita anche con il termine ticket dall'analisi del codice e dalle verifiche effettuate può avere diverse lunghezze (40, 56, 150, 166 byte).

Session Key

Una chiave di cifratura temporanea impiegata fra due principal, con un tempo di vita limitato alla durata di un relativo ticket. Nei codici sorgenti esaminati viene anche denominata Handshake Key.

Shared Secret

Informazione conosciuta solo dalle parti che si stanno mutuamente autenticando; esse potrebbero venire a conoscenza tramite una terza parte considerata fidata dalle prime due. Dovrebbe essere costituita dalla Session Key.

TGS (Ticket-Granting Service)

La parte del KDC che distribuisce i ticket ai client per gli specifici servizi. Il processo utente comunica con il TGS grazie ad un TGT.

TGT (Ticket-Granting Ticket)

Uno speciale ticket Kerberos che consente al client di ottenere, in maniera trasparente altri ticket.

Ticket

Un insieme di credenziali elettroniche che verifica l'identità di un client per la fruizione di uno specifico servizio.

Token

Struttura impiegata da AFS per l'autorizzazione e l'accesso al file system. Al suo interno contiene un ticket. Nei codici esaminati è suddivisa in due sezioni fondamentali il Secret Token (anche riferito con il termine Ticket) ed il Clear Token.

Vice

In ambiente AFS indica l'insieme dei server ed il termine deriva dal nome del software che gira su di essi. Il termine è anche usato come suffisso per l'identificazione, nei codici sorgenti, di strutture e dati (ViceIOCTL, ViceID).

2. Contesto, obiettivi e scopo

Il Progetto CRESCO: Centro computazionale di RicErca sui Sistemi Complessi ha come obiettivo la realizzazione, presso il Centro Ricerche ENEA di Portici (NA), di un importante Polo di calcolo multidisciplinare per lo studio dei sistemi complessi di natura biologica e tecnologica, collegato con *facilities* di elaborazione, archiviazione e visualizzazione 3D di dati ubicate presso i Centri ENEA di Brindisi e Trisaia (MT).

Sul piano delle applicazioni e dei contenuti scientifici, il Progetto si focalizza sulle seguenti linee d'attività:

- Linea di Attività 1 (LA1): Realizzazione del Polo di Calcolo e sviluppo di nuove funzionalità di GRID Computing
l'implementazione di soluzioni innovative in tema di architetture di sistemi di calcolo e di GRID computing per le attività di R&S di punta dell'ENEA che richiedano l'utilizzo di risorse computazionali estremamente importanti. Sul versante infrastrutturale il Progetto prevede la realizzazione di una piattaforma di calcolo di assoluta eccellenza in grado di erogare una potenza dell'ordine del Teraflop, corredata di una nuova classe di funzionalità GRID innovative;
- Linea di Attività 2 (LA2): Sviluppo di tecnologie e modelli computazionali per la descrizione di sistemi complessi di origine biologica e di materiali innovativi
lo studio di oggetti biologici dal punto di vista "sistemico" (in silico cell) e lo studio di sistemi naturali (comunità animali e sociali) secondo il paradigma dei sistemi complessi;
- Linea di Attività 3 (LA3): Sviluppo di modelli di simulazione ed analisi delle Reti tecnologiche complesse e delle loro interdipendenze
lo studio di sistemi tecnologici complessi e delle loro mutue interazioni, e la realizzazione di opportuni strumenti per il modelling, la simulazione e il controllo.

Nell'ambito delle attività previste dalla Linea di Attività 1 il sottoprogetto identificato come SP I.2, *Sviluppo ed integrazione dell'architettura GRID e di grafica 3D*, prevede, fra l'altro, l'attività di integrazione dell'ENEA-GRID con altre architetture di *GRID computing* e nell'ambito di tale attività, nella quale il Dipartimento di Ingegneria dell'Informazione e Matematica Applicata dell'Università degli Studi di Salerno è coinvolto, è prevista, primariamente, un'analisi dell'infrastruttura ENEA-GRID [1] con

l'individuazione dei punti critici ai fini dell'interoperabilità, in particolar modo con sistemi Windows, e dell'estensione delle caratteristiche di gestione e controllo dell'infrastruttura stessa.

Uno dei problemi emersi dal documento di analisi è quello relativo al non supportato trasferimento del token da parte del gestore delle risorse LSF dal mondo Unix a quello Windows. In effetti LSF non supporta AFS in ambiente Windows e quindi, nel momento in cui si desidera attivare un job su una macchina Windows, il mancato trasferimento del token AFS (che ovviamente consente all'utente di poter accedere al proprio volume) determina di fatto l'impossibilità di avere il proprio spazio a disposizione sull'host Windows e, nel caso il sistema di autorizzazione risultasse basato sulle ACL di AFS, l'operazione sarebbe di fatto impossibile. È dunque opportuno verificare se sia possibile effettuare l'operazione di trasferimento del token da e verso sistemi Windows, partendo dai codici sorgenti delle routine a ciò destinate e disponibili per il solo mondo Unix.

Lo studio di fattibilità per l'operazione del *porting* in ambiente Windows del trasferimento del token AFS per LSF si muove lungo due direzioni:

1. studio ed analisi del kit che serve a LSF per trattare i token AFS denominato *Platform LSF Source Code Package for AFS*, disponibile per il solo mondo Unix, e nel seguito riferito semplicemente come kit LSF;
2. studio ed analisi dei sorgenti di OpenAFS per Windows per stabilire dei punti in comune fra i due ambienti che potrebbero essere utili per l'operazione di *porting*.

Prima di presentare i risultati relativi alle linee di azione sopra citate è opportuno premettere un paragrafo relativo alle API per AFS la cui corretta comprensione è fondamentale per esaminare sia le routine del kit LSF sia di quelle presenti nei sorgenti di OpenAFS per Windows.

3. Le API per AFS

Anche se la maggior parte delle operazioni sui file archiviati in AFS può essere effettuata impiegando le *system call* standard di Unix, alcune operazioni, specifiche del *file system* distribuito, non hanno equivalenti Unix.

E' chiara quindi l'esigenza di *system call* che consentano di applicare le speciali funzioni previste per i file archiviati nell' *Andrew File System*, e che non avrebbero senso se applicate a file memorizzati su un *file system* Unix. Operazioni di tal tipo sono ad esempio quelle relative alle *Access Control List* associate alle directory, quelle concernenti il sistema di quote associato con i volumi AFS e anche quelle relative alla gestione dei token di cui ci occuperemo in dettaglio nel prosieguo.

Quindi per la programmazione esistono due interfacce verso AFS[4]. La prima impiega la *system call* standard `ioctl` mentre la seconda ne usa una nuova denominata `pioctl`. La `pioctl` usa gli stessi parametri della *system call* `ioctl` tranne per il fatto che il primo parametro nella `pioctl` è un *pathname* mentre nella `ioctl` è un descrittore di file.

Nell'invocazione di tale *system call*, uno dei quattro parametri previsti è una struttura definita `struct ViceIoctl`. Tale struttura risulta composta da due puntatori (in e out) e da due short (`in_size` e `out_size`) ed in generale i dati sono passati ad AFS nel buffer di input (puntato dal puntatore in e che contiene `in_size` byte) ed i risultati sono presi nel buffer di output (puntato dal puntatore out e che contiene `out_size` byte). Nel kit LSF la struttura è definita nel file `afslib.h` come:

```
struct ViceIoctl {
    caddr_t in, out;
    short in_size;
    short out_size;
};
```

Mentre per Windows, ed in particolare per OpenAFS, è definita nel file `pioctl_nt.c` come:

```
struct ViceIoctl {
    long in_size;
    long out_size;
    void *in;
    void *out;
};
```

Il tipo di operazione che la *system call* deve svolgere viene specificato al momento dell'invocazione tramite un valore costante (valori di esempio `VIOCGETTOK`, `VIOCSETTOK`, `VIOCGETAL`, `VIOCSETAL`). Anche se la scelta è sconsigliata¹, è possibile trovare tali valori definiti in funzione di un'altra macro `_VICEIOCTL`

¹ Come si vedrà nel seguito tale approccio è peraltro impiegato anche nel Platform LSF Source Code Package for AFS

valorizzata mediante la specifica di un indice (ad esempio VIOCGETTOK diventa _VICEIOCTL(3) e VIOCSETTOK diventa _VICEIOCTL(8))

Per una completa lista di questi indici consultare il § 3.2 dell'appendice A.

A titolo esemplificativo riportiamo l'invocazione di una `pioctl` dettagliando il significato dei quattro parametri previsti. Il primo è detto *pathname parameter* ed è il nome del file interessato dall'operazione, il secondo parametro indica l'operazione tramite uno delle costanti sopra citate, il terzo è il puntatore alla struttura `ViceIoctl` ed il quarto determina quale file debba essere impiegato nel caso il *pathname* indichi un link simbolico. Nei casi in cui la `pioctl` non sia relativa ad un file il parametro del *pathname* deve essere NULL (0). Il seguente esempio di `pioctl` setta un ACL relativa a `/afs/andrew/usr/kazar` per consentire a tutti di leggere la directory.

```
struct ViceIoctl blob;  
char *acl = "1\n0\nSystem:AnyUser      127\n";  
blob.in_size = strlen(acl)+1; /* don't forget the null */  
blob.in = acl;  
blob.out_size = 0; /* no out parameters */  
code = pioctl("/afs/andrew/usr/kazar", VIOCSETAL, &blob, 1);  
if (code < 0) perror("setacl");
```

Un documento di maggior dettaglio relativo alle API di AFS è riportato per completezza e per riferimenti ulteriori in appendice A.

4. Platform LSF Source Code Package for AFS

Per fare in modo che LSF supporti il *file system* AFS è necessario installare il *package* per AFS fornito dalla Platform. Partendo dai sorgenti, nativi per il mondo Unix/Linux, è possibile costruire gli eseguibili che dovranno essere opportunamente copiati nella directory di LSF (individuata attraverso la variabile `$LSF_SERVERDIR`²).

Nello specifico i file che dovranno essere copiati dopo il processo di *build* sono `daemon.wrap`, `gettok`, `puttok`, `rtok`, `esub`, and `eexec`.

Nel seguito saranno esaminati gli scopi e le caratteristiche di ogni singolo file.

4.1. Logica di impiego delle routine del package

Il supporto per AFS da parte di LSF è realizzato mediante i meccanismi di sottomissione e di controllo costituiti dagli eseguibili `esub` ed `eexec` [5].

LSF può validare, modificare o rifiutare la sottomissione di un job, modificarne il suo ambiente di esecuzione e passare dati direttamente dall'host di sottomissione a quello di esecuzione mediante i sopra citati meccanismi. Sia `esub` che `eexec` devono essere memorizzati nella directory puntata da `LSF_SERVERDIR`. Nello specifico:

- per valicare modificare o rifiutare un job si deve scrivere un meccanismo `esub`;
- per modificare l'ambiente di esecuzione sull'host di esecuzione si deve scrivere un meccanismo `eexec`;
- per passare i dati direttamente all'host di esecuzione devono essere impiegati i due meccanismi di `esub` ed `eexec`. In effetti, il caso dell'inoltro del token AFS all'host di esecuzione viene proprio realizzato tramite i due meccanismi.

4.1.1. Il metodo `esub`

Il metodo `esub`, abbreviazione per *external submission*, è un eseguibile redatto dall'utente (binario o script) che può essere utilizzato per valicare, modificare o rifiutare i jobs. L'eseguibile `esub` deve essere archiviato nella `LSF_SERVERDIR` (definita nel file di configurazione in `lsf.conf`) dove LSF ne verifica l'esistenza quando un job viene sottomesso, riavviato o modificato. Se LSF trova tale `esub`, l'eseguibile è

² `LSF_SERVERDIR`, definita nel file di configurazione `lsf.conf`, è la directory per tutti i file binari e gli script del server e gli eseguibili esterni invocati dai demoni LSF, deve essere di proprietà di root e condivisa da tutti gli host dello stesso tipo

eseguito da LSF determinando, in base alla logica presente nell'*esub* stesso, se il job viene sottomesso, modificato o rifiutato. Ogni messaggio che deve essere fornito all'utente dovrebbe essere rediretto verso lo standard error (*stderr*) e non verso lo standard output (*stdout*).

Nell'ambiente di esecuzione dell'*esub* sono disponibili una serie di variabili di ambiente che determinano il flusso di esecuzione del codice stesso nonché le azioni di LSF.

Se il valore di uscita dell'*esub* è dato da *LSB_SUB_ABORT_VALUE* allora LSF scarta il job, in caso diverso viene valutata l'esistenza delle variabili *LSB_SUB_MODIFY_FILE* or *LSB_SUB_MODIFY_ENVFILE* ed in caso positivo LSF applica i cambiamenti apportati.

In effetti, LSF può applicare in tre modi il meccanismo *esub* ed il processo di esecuzione dei vari eseguibili *esub* è gestito dal processo *mesub* (master *esub* in *LSF_SERVERDIR*).

Quando *bsub* sottomette un job lancia *mesub* per gestire l'eventuale scelta ed esecuzione degli *esub*.

1. È possibile specificare un *mandatory esub* nel file di configurazione *lsf.conf* mediante la specifica della variabile *LSB_ESUB_METHOD* che dovrà specificare il nome del metodo *esub*, ove per metodo *esub* si intende l'eseguibile (presente in *LSF_SERVERDIR*) avente nome *esub.nomemetodo*;
2. Un qualsiasi eseguibile denominato *esub* presente in *LSF_SERVERDIR*;
3. LSF fornisce un master *esub* (*LSF_SERVERDIR/mesub*) per gestire le invocazioni dei diversi specifici *esub* create dagli utenti. L'impiego dell'opzione *-a* di *bsub* consente di specificare quale specifico *esub* impiegare da LSF. Ad esempio mediante il comando:

bsub -a fluent bsub_options fluent_command

LSF applica prima di eseguire il comando *fluent*, l'eseguibile *esub.fluent* presente al solito in *LSF_SERVERDIR*.

È importante notare che i vari metodi *esub* che verranno applicati lo saranno nell'ordine sopra indicato (quindi prima il *mandatory*, poi l'*esub* presente in *LSF_SERVERDIR* e poi quelli specificati nel parametro *-a* della *bsub*).

Un elenco completo delle variabili disponibili nell'ambiente di esecuzione dell'*esub* è riportato in Appendice B.

4.1.2. Il metodo `eexec`

L'eseguibile `eexec` gira sull'host di esecuzione all'avvio del job, al suo termine e quando viene iniziato un *check point*. E' eseguito come processo utente dopo che le variabili di ambiente del job sono state settate. La variabile di ambiente `LS_EXEC_T` è settata a `START`, `END`, e `CHKPNT`, per indicare il momento in cui `eexec` è invocato.

Ci si aspetta che `eexec` termini in quanto il processo job padre aspetta `eexec` per completare la sua esecuzione. L'ID del processo che invoca `eexec` è settato nella variabile di ambiente `LS_JOBPID`. Nel caso lo scopo di `eexec` sia quello di monitorare l'esecuzione del job, `eexec` deve effettuare il *fork* di un figlio e poi terminare. Il processo `eexec` figlio dovrà poi periodicamente verificare se il job sia ancora in esecuzione impiegando la suddetta variabile `LS_JOBPID`.

L'impiego congiunto di `esub` ed `eexec` consente di passare i dati all'ambiente di esecuzione. In effetti, se `esub` deve passare dei dati a `eexec`, può scrivere tali dati sul suo standard output e gli stessi verranno letti sullo standard input di `eexec` in quanto LSF agisce da *pipe* fra `esub` ed `eexec`. (`esub | eexec`). Quindi lo `stdout` di ogni metodo `esub` è automaticamente inviato a `eexec`. Poiché `eexec` non può trattare più di un *output stream* sarà possibile usare un solo `esub` e quindi un solo standard output per generare i dati sullo standard input di `eexec`.

Tale meccanismo è alla base dell'inoltro del token per AFS. In tal caso avremo un metodo `esub` specifico per AFS (ad esempio `esub.afs`) che invia il suo token di autenticazione tramite lo standard output a `eexec` il quale provvederà ad attivarlo sull'host di sottomissione.

4.2. La routine gettok

Nel kit che Platform mette a disposizione per sfruttare i token AFS è presente, fra l'altro, la routine `gettok` sviluppata da LSF con lo scopo di stampare sullo *standard output* le informazioni relative al token AFS attualmente in uso sul sistema. La routine ha un ruolo chiave nell'operazione di trasferimento del token da parte di LSF, che consente, all'utente che lancia il job, di avere a disposizione sulla macchina che lo eseguirà il proprio *file system*.

La routine non è disponibile per l'ambiente Windows in quanto Platform conclama la compatibilità del trasferimento del token solo per alcuni ben definiti Unix.

L'obiettivo è quindi quello di esaminare approfonditamente tale routine per valutare la possibilità di realizzarne una versione che possa funzionare sotto sistemi operativi Windows.

Il codice della routine `gettok` è di seguito riportato:

```
/* *****  
 * gettok.c --  
 * This file contains load sharing routines that are related  
 * to getting AFS tokens.  
 * *****  
#include "afslib.h"  
  
extern void printToken (AFSTOKEN);  
extern int getAFSTokens(AFSTOKENS *);  
  
/* main */  
int  
main (int argc, char **argv)  
{  
    AFSTOKENS afsTokens;  
    int getTime = FALSE, endTime = INFINIT_INT;  
    int i, lastTime;  
    time_t now;  
  
    now = time(0);  
  
    if (argv[1] && strcmp (argv[1], "-t") == 0)  
        getTime = TRUE;  
  
    /* Prelievo del/dei token/s */  
    if (getAFSTokens(&afsTokens) == -1) {  
        fprintf(stderr, "getAFSTokens() failed\n");  
        exit (-1);  
    }  
  
    /* Stampa su standard output del/dei token/s */  
    if (getTime == FALSE && afsTokens.num > 0) {  
        fprintf (stdout, "%d ", afsTokens.num);  
    }  
    for (i = 0; i < afsTokens.num; i++) {  
        if (getTime == FALSE)  
            printToken (afsTokens.tok[i]);  
        if (AFS_TOKEN_CLEAR(afsTokens.tok[i].data)->EndTimestamp < endTime)  
            endTime = AFS_TOKEN_CLEAR(afsTokens.tok[i].data)->EndTimestamp;  
    }  
  
    if (getTime) {
```

```

    if (afsTokens.num == 0) /* no tokens */
        lastTime = INFINIT_INT;
    else {
        if ((lastTime = endTime - now) < 0)
            lastTime = 0;
    }
    fprintf (stdout, "%d \n", lastTime);
}
exit (0);
} /* main */

```

Le due principali operazioni svolte dalla routine sono effettuate dalle routine `getAFSTokens` e `printToken`, definite nel file di libreria `afslib.c`. La prima routine restituisce una struttura che consente di accedere fino ad un massimo di cinque token presenti sul sistema, mentre la seconda provvede alla loro visualizzazione sullo *standard output*.

Nel corpo della routine `getAFSTokens` sono impiegate le *system call* specifiche di AFS per ottenere il token.

4.2.1. La routine `getAFSTokens`

Alla luce di questo esaminiamo il codice della routine `getAFSTokens` che restituisce una struttura di tipo `AFSTOKENS` mediante la quale è possibile accedere, attraverso una lista di strutture di tipo `AFSTOKEN`, ai token presenti nel sistema.

I token saranno letti tramite la *system call* `pioctl`.

La *Figura 1 - Strutture dati coinvolte nella routine `getAFSTokens`* riporta le principali strutture dati impiegate nella routine.

```
/* getAFSTokens */
```

```

Int getAFSTokens(AFSTOKENS3 *tokens)
{
    struct ViceIoctl iob;
    char tokBuf[AFS_TOKEN_LEN4];

    /* We will allocate 5 tokens at a time */
    /* ----- VEDI PASSO 1 ----- */
    if ((tokens->tok = (AFSTOKEN *) calloc(5, sizeof(AFSTOKEN))) == NULL)
        return(-1);

    /* We only break out of the loop using return() */
    /* ----- VEDI PASSO 2 ----- */
}

```

³ In `afslib.h` troviamo le definizioni:

```

typedef struct _myafsToken {
    int      len;
    char     *data;
} AFSTOKEN;
typedef struct _myafsTokens {
    int      num;
    AFSTOKEN *tok;
} AFSTOKENS;

```

⁴ In `afslib.h` troviamo le definizioni:

```

/* This should be enough since the cellname can only be 256 */
#define AFS_TOKEN_LEN 512

```

```

    for (tokens->num=0; ;) {

        /* Inizializzazione delle strutture token */
        INIT_AFS_TOKEN5(&(tokens->tok[tokens->num]));

        /* We will try to get as many tokens as possible. */
        if ((tokens->num + 1) % 5 == 0) {
            AFSTOKEN *t;

            /* Se si superano i 5 token si effettua una realloc */
            if ((t = (AFSTOKEN *) realloc(tokens->tok, (tokens->num + 6) *
sizeof(AFSTOKEN))) == NULL) {
                return (0);
            }
            tokens->tok = t;
        }

        /* Settaggio della struttra ViceIoctl da passare alla funzione pioctl */
        /* ----- VEDI PASSO 3 ----- */
        iob.in = (caddr_t)&tokens->num; /* numero di token */
        iob.in_size = sizeof(tokens->num); /* dimensione del campo iob.in */
        iob.out = tokBuf; /* buffer che conterrà il token dopo la pioctl */
        iob.out_size = AFS_TOKEN_LEN; /* size di tokbuf */

        /* ----- VEDI PASSO 4 ----- */
        if (pioctl(0, _VICEIOCTL(8), &iob, 0)) {
            if (errno == EDOM6) {
                if (tokens->num == 0) { /* Non sono stati trovati tokens */
                    freeAFSTokens(tokens); /* deallocazione spazio del token (afslib.c) */
                }
                return (0); /* Esco dalla routine perché non ci sono token */
            }
            freeAFSTokens(tokens);
            return (-1); /* Esco dalla routine perché pioctl fallisce */
        }

        /* Metto la lunghezza del token */
        if ((tokens->tok[tokens->num].len = lenAFSToken(tokBuf)) == -1)7 {
            /* skip oversize token */
            /* VEDI PASSO 5 */
            continue;
        }

        /* Alloco spazio per il token */
        /* ----- VEDI PASSO 6 ----- */
        if ((tokens->tok[tokens->num].data = (char *)
calloc(tokens->tok[tokens->num].len, sizeof(char))) == NULL) {
            if (tokens->num == 0)
                freeAFSTokens(tokens);
            return (0);
        }

        /* Copio da tokbuf nella zona allocata */
        /* ----- VEDI PASSO 7 ----- */
        memcpy(tokens->tok[tokens->num].data, tokBuf, tokens->tok[tokens->num].len);
        tokens->num++;
    }

```

⁵ In afslib.h troviamo le definizioni:

```

/* Initialize the struct afstoken fields */
#ifdef INIT_AFS_TOKEN
#define INIT_AFS_TOKEN(t) {(t)->len = 0; (t)->data = NULL;}
#endif

```

⁶ Il codice di errore EDOM viene dato quando non ci sono più token

⁷ Nello stesso file afslib.c troviamo la definizione delle funzione

```

* lenAFSToken
* Returns length of token if it is less than MAX_AFS_TOKEN_LEN,
* Otherwise return -1

```

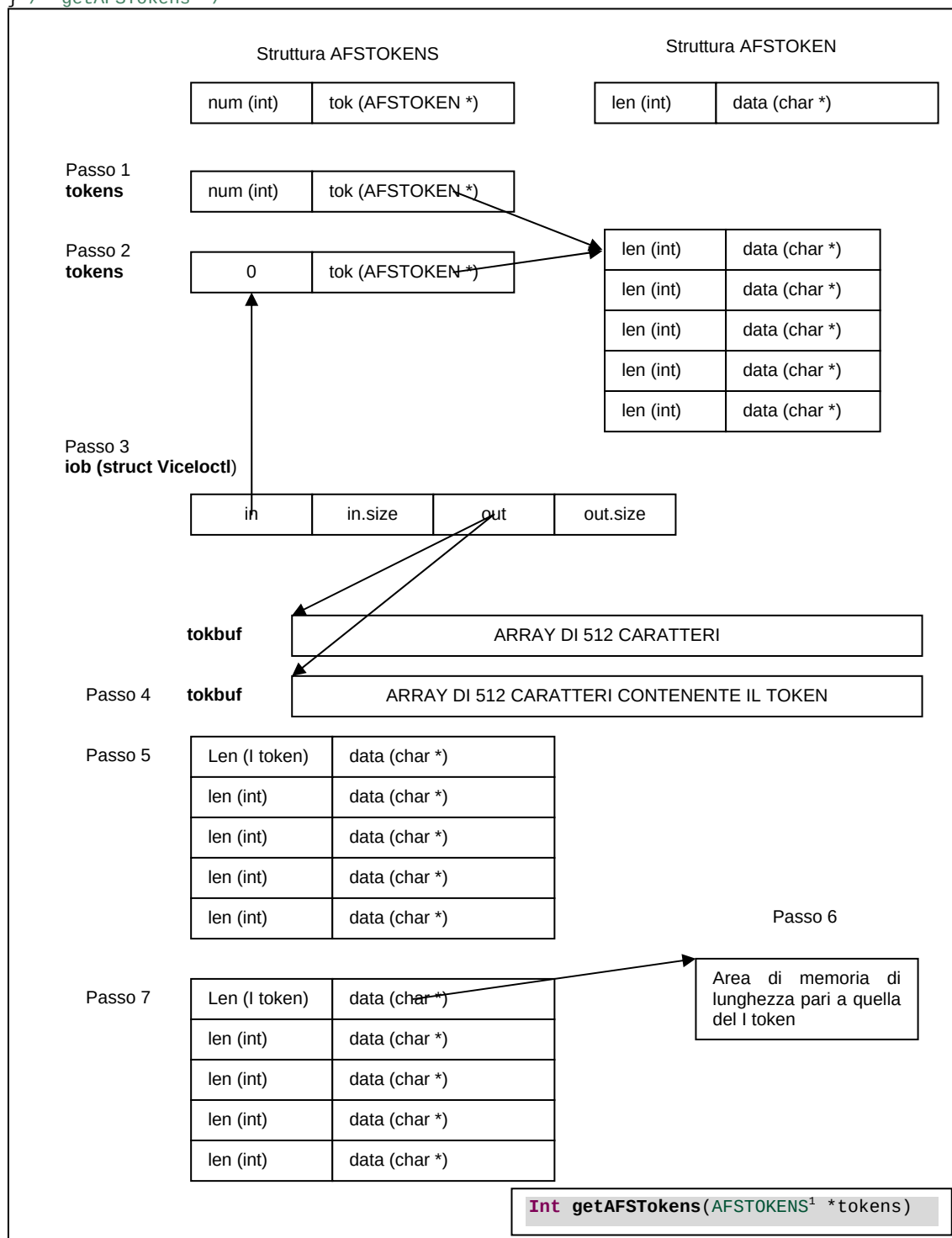
```

}

fprintf(stderr, "getAFSTokens_: Should never reach!!!\n");
return (-1);

} /* getAFSTokens */

```

Figura 1 - Strutture dati coinvolte nella routine `getAFSTokens`

4.2.2. La routine printToken

Esaminiamo ora la funzione `printToken` e, per una sua corretta comprensione, bisogna conoscere il contenuto del token prelevato tramite la routine esaminata in precedenza.

In effetti i token viaggiano in coppia; un **clear token** e un **secret token**. Il *secret token* è cifrato dall'*Authentication Server* con la chiave del *file server* e, in quanto cifrato, può essere trasmesso in sicurezza attraverso la rete. Il *clear token* contiene le stesse informazioni del *secret token*, ma in chiaro e quindi non dovrebbe essere trasmesso sulla rete.

La routine `printToken` si occupa della stampa di un solo token e, poiché la `getAFSTokens` ne tratta fino a un massimo di 5, essa sarà invocata iterativamente nell'ambito della `gettok`.

Un singolo token, memorizzato in `afsTokens.tok[ ].data`, letto da `getAFSTokens` mediante la *system call* `pioctl`, può essere così suddiviso:

- i primi 4 byte danno la dimensione del secret token;
- i successivi `sizeof(struct SecretToken)` byte contengono il secret token;
- i successive 4 byte danno la dimensione del clear token;
- i successivi `sizeof(struct ClearToken)` byte contengono il clear token;
- i successivi 4 byte rappresentano un identificativo della cella;
- il nome della cella.

La struttura `ClearToken` è a sua volta suddivisa in una serie di campi, mentre il *secret token*, essendo cifrato, non ha nessuna modalità interpretativa. Il *clear token* risulta composto da:

- | | |
|--------------------------------------|---|
| • <code>AuthHandle (int)</code> | numero di versione della chiave di sessione |
| • <code>HandShakeKey (char 8)</code> | chiave di sessione |
| • <code>ViceId (int)</code> | identificativo utente |
| • <code>BeginTimestamp (int)</code> | inizio validità del token |
| • <code>EndTimestamp (int)</code> | fine validità del token |

La struttura è definita in `afslib.h` come

```
struct ClearToken {
    int AuthHandle;
    char HandShakeKey[8];
    int ViceId;
    int BeginTimestamp;
    int EndTimestamp;
};
```

All'interno della `printToken` l'accesso alle varie sezioni del token avviene mediante le seguenti definizioni che troviamo in `afslib.h`

```
/* Fields of afsTokens.tok[ ].data */
#define AFS_TOKEN_SECRET_LEN(t) (*(int *) (t))
#define AFS_TOKEN_SECRET(t) ((char *) (&(AFS_TOKEN_SECRET_LEN(t)) + 1))
```

```

#define AFS_TOKEN_CLEAR_LEN(t)((int *) (AFS_TOKEN_SECRET(t) + AFS_TOKEN_SECRET_LEN(t)))
#ifndef AFS_TOKEN_CLEAR
#define AFS_TOKEN_CLEAR(t)((CLEARTOKEN *) (&(AFS_TOKEN_CLEAR_LEN(t)) + 1))
#endif
#define AFS_TOKEN_CELL(t) ((int *))((char *) AFS_TOKEN_CLEAR(t)+AFS_TOKEN_CLEAR_LEN(t))
#define AFS_TOKEN_CELLNAME(t) ((char *) (&(AFS_TOKEN_CELL(t)) + 1)

```

Di seguito il codice della printToken mentre in *Figura 2 - Strutture dati coinvolte nella routine printTokens* sono riportate le principali strutture dati coinvolte.

```

/* printToken */
void printToken (AFSTOKEN tok)
{
    char *buf;
    int i, a, len;
    u_int b;

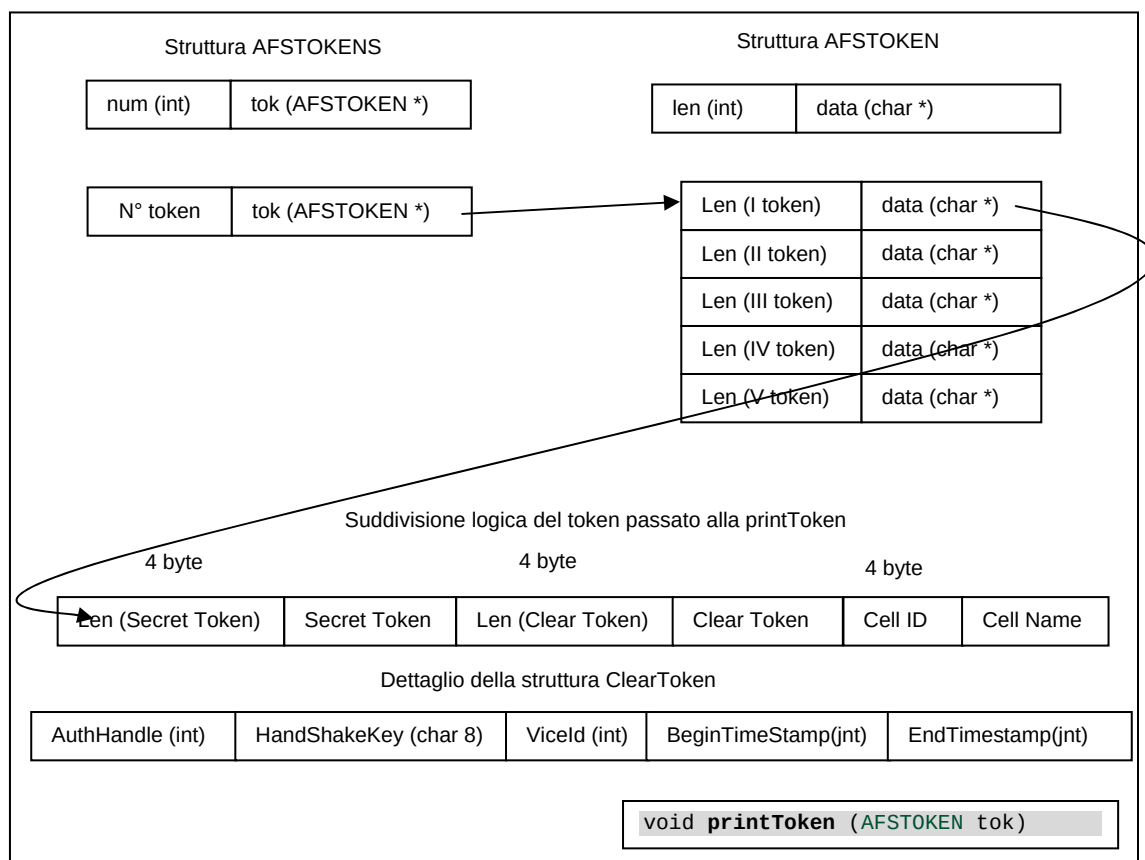
    buf = tok.data;
    printf("%d ", tok.len); /* Lunghezza totale della struttura */
    printf("%d ", AFS_TOKEN_SECRET_LEN(buf)); /* Lunghezza del secret token*/

    /* Stampa il secret token */
    for (i=0; i<AFS_TOKEN_SECRET_LEN(buf); i++) {
        a = AFS_TOKEN_SECRET(buf)[i];
        printf("%x ", AFS_TOKEN_SECRET(buf)[i]);
    }

    printf ("%d ", AFS_TOKEN_CLEAR_LEN(buf)); /* Dimensione del ClearToken*/
    len = sizeof(AFS_TOKEN_CLEAR(tok.data)->HandShakeKey); /* Lunghezza di HandShakeKey*/
    printf ("%d ", len);

    /* Stampa HandShakeKey del ClearToken */
    for (i = 0; i < len; i++) {
        b = AFS_TOKEN_CLEAR(tok.data)->HandShakeKey[i];
        printf ("%d ", b);
    }
    printf ("%d %d %d %d %s ",
        AFS_TOKEN_CLEAR(buf)->AuthHandle,
        AFS_TOKEN_CLEAR(buf)->ViceId,
        AFS_TOKEN_CLEAR(buf)->BeginTimestamp,
        AFS_TOKEN_CLEAR(buf)->EndTimestamp,
        AFS_TOKEN_CELLNAME(buf));
} /* printToken */

```

**Figura 2 - Strutture dati coinvolte nella routine printTokens**

Siamo in possesso di tutti gli elementi necessari per poter esaminare l'output della routine `gettok`. La routine è stata eseguita su un sistema della GRID ENEA, precisamente su `kleos.portici.enea.it`.

```
/afs/enea.it/por/user/romano> gettok
```

```
1 100 56 18 1e cf 6a af 6e e3 26 cc b3 cb 48 d3 f e5 bb 2b 6 8c 9 18 21 de 4c 2c 2e a3
47 72 bc 22 e9 49 52 2f 62 69 6e 2f 6c 6d 68 6f 73 74 69 64 20 3b 20 7d a 0 0 1 0 24 8
174 227 42 220 247 193 50 239 0 26148 1191408562 1194000561 enea.it
```

Suddividiamo l'output in blocchi per comprendere meglio come è strutturato.

Il primo valore (1) rappresenta il numero di token presenti nel sistema

Il secondo (100) rappresenta la lunghezza del token

Il terzo valore (56) rappresenta la lunghezza del *secret token*

I 56 numeri in rosso rappresentano il *secret token*

Il valore successivo (24) è la dimensione della struttura `ClearToken`

Il valore seguente (8) è la dimensione dell'`HandShakeKey`

Gli 8 numeri in verde rappresentano `HandShakeKey`

`AuthHandle` (0)

`Viceld` (26148)

`BeginTimeStamp` (111408562)

`EndTimeStamp` (1194000561)

Nome della Cella (enea.it).

4.3. La routine puttok

Il compito della routine puttok è quello di scrivere nel kernel i token AFS letti dallo *standard input*.

L'esamina del codice consente di focalizzare l'attenzione sui seguenti punti chiave:

1. Per svolgere tale attività impiega la `converAFSTokens` che legge i dati e restituisce una struttura `AFSTOKENS`, poi la `putAFSTokens` si occupa di scrivere i token nel kernel.
2. La routine `puttok`, inoltre, se invocata con il parametro `-d`, visualizza anche, tramite la già esaminata routine `printToken`, i token trattati.
3. La `converAFSTokens` è presente nel file `puttok.c`, mentre la `putAFSTokens` è definita in `afslib.c`.

4.3.1. La routine `converAFSTokens`

La routine `converAFSTokens` è invocata con parametro `tokens`, struttura di tipo `AFSTOKENS` e per la precisione gli viene passato l'indirizzo della variabile in modo che alla fine della procedura, `tokens` sia una struttura per il contenimento dei token come quella già vista per `gettok`.

Quindi la `puttok` chiama `converAFSTokens` per ottenere la struttura `tokens`, così come risulta in *Figura 3 Strutture dati coinvolte nella routine `converAFSTokens`*, prendendo i dati dallo *standard input*.

```

/*****
 * puttok.c --
 *
 * This file contains load sharing routines that are related
 * to putting AFS tokens.
 *****/

#include "afslib.h"

extern void printToken (AFSTOKEN);
static int converAFStokens (AFSTOKENS *);
extern int putAFSTokens (AFSTOKENS *);

/* main */
int
main (int argc, char **argv)
{
    AFSTOKENS tokens;
    int i;

    if (converAFStokens(&tokens) < 0) {
        fprintf (stderr, "Tokens not converted\n");
        exit (2);
    }

    if (putAFSTokens(&tokens) == -1)
        exit(-1);

    if (argc > 1 && !strcmp(argv[1], "-d")) {
        for (i = 0; i < tokens.num; i++)
            printToken(tokens.tok[i]);
    }

    exit (0);
} /* main */

/* convertAFStokens */
static int converAFStokens (AFSTOKENS *tokens)
{
    int i, len, cc, j, a, b, c, d, numTokens;

    tokens->num = 0;
    /* Legge da standard input il numero di token */
    if ((cc = fscanf(stdin, "%d", &numTokens)) != 1) {
        fprintf(stderr, "Fail to get numTokens\n");
        return (-1);
    }

    /* Allocations spazio per il numero di strutture token richiesto */
    if ((tokens->tok = (AFSTOKEN *) calloc (numTokens, sizeof (AFSTOKEN))) == NULL) {
        fprintf(stderr, "Malloc() failed, numTokens=%d: ", numTokens);
        perror("");
        return (-1);
    }

    /* ----- VEDI PASSO 1 ----- */
    for (i = 0; i < numTokens; i++) {
        /* Inizializzazione dell'i-esima struttura tokens */
        INIT_AFS_TOKEN(&(tokens->tok[i]));

        /* Lunghezza del token */
        if ((cc = fscanf (stdin, "%d", &tokens->tok[i].len)) != 1) {
            fprintf (stderr, "Fail to get length of token %d\n", i);
            return (-1);
        }

        /* Allocations spazio per le strutture token */
        if ((tokens->tok[i].data = (char *) calloc(tokens->tok[i].len,
            sizeof(char))) == NULL) {

```

```

        fprintf(stderr, "calloc() failed for token %d len %d: ",
                    i, tokens->tok[i].len);
        perror("");
        return (-1);
    }

    /* ----- VEDI PASSO 2 ----- */
    if ((cc = fscanf (stdin, "%d", &len)) != 1) {
        /* token's secret length */
        fprintf (stderr, "Failed to get secret length of token %d\n", i);
        return (-1);
    }
    AFS_TOKEN_SECRET_LEN(tokens->tok[i].data) = (char) len;

    /* Legge da standard input il secret token */
    for (j = 0; j < len; j++) {
        if ((cc = fscanf (stdin, "%x", &a)) != 1) {
            fprintf (stderr,
                    "Failed to get the secret part for token %d\n", i);
            return (-1);
        }
        AFS_TOKEN_SECRET(tokens->tok[i].data)[j] = (char)a;
    }

    /* Legge da input la dimensione struttura secret token */
    if ((cc = fscanf (stdin, "%d", &len)) != 1) {
        fprintf(stderr,
            "Failed to get the clear length of token %d\n", i);
        return (-1);
    }
    AFS_TOKEN_CLEAR_LEN(tokens->tok[i].data) = (char) len;

    /* Controllo che la dimensione letta da standard input sia
     * uguale alla effettiva dimensione di una struttura ClearToken */
    if (AFS_TOKEN_CLEAR_LEN(tokens->tok[i].data) != sizeof(CLEARTOKEN)) {
        fprintf(stderr, "Token %d: clear length %d not equal to %d\n",
            i, AFS_TOKEN_CLEAR_LEN(tokens->tok[i].data),
            sizeof(CLEARTOKEN));
        return (-1);
    }

    /* Legge da input la lunghezza di HandShakeKey */
    if ((cc = fscanf (stdin, "%d", &len)) != 1) {
        fprintf (stderr, "Failed to get handShakeKey size for token %d\n",
            i);
        return (-1);
    }

    /* Legge da input l'HandShakeKey */
    for (j = 0; j < len; j++) {
        if ((cc = fscanf (stdin, "%d", &a)) != 1) {
            fprintf (stderr,
                    "Failed to get handhakeKey %d for token %d\n", j, i);
            return (-1);
        }
        AFS_TOKEN_CLEAR(tokens->tok[i].data)->HandShakeKey[j] = (char)a;
    }

    /* Legge nell'ordine AuthHandle, ViceId,
     * BeginTimestamp, HandTimestamp e CellName */
    if ((cc = fscanf (stdin, "%d%d%d%d%s", &a, &b, &c, &d,
        AFS_TOKEN_CELLNAME(tokens->tok[i].data))) != 5) {
        fprintf (stderr, "Failed to get data for token %d\n", i);
        return (-1);
    }
    AFS_TOKEN_CLEAR(tokens->tok[i].data)->AuthHandle = a;
    AFS_TOKEN_CLEAR(tokens->tok[i].data)->ViceId = b;
    AFS_TOKEN_CLEAR(tokens->tok[i].data)->BeginTimestamp = c;
    AFS_TOKEN_CLEAR(tokens->tok[i].data)->EndTimestamp = d;
}

```

```

tokens->num = numTokens;
return (0);
}

```

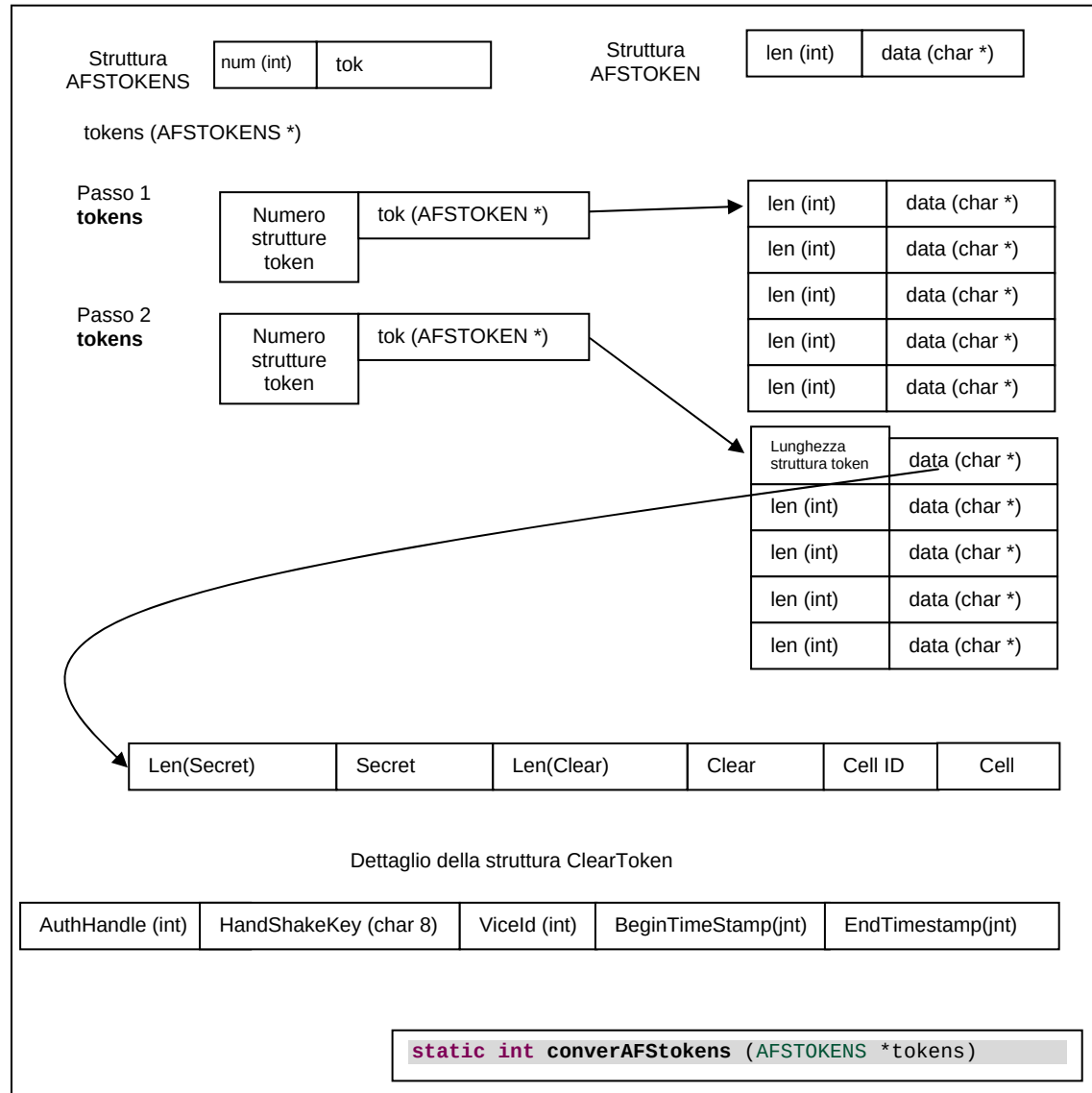


Figura 3 Strutture dati coinvolte nella routine converAFSTokens

4.3.2. La routine putAFSTokens

La putAFSTokens, definita in afslib.c, ha il compito di scrivere i token, in precedenza prelevati dalla converAFSTokens, nel kernel.

Come si nota dal sorgente riportato, la routine impiega pioctl per scrivere il token; la *system call* è chiamata iterativamente per scrivere tutti i token presenti nella struttura

```
/*
 * putAFSTokens(tokens)
 *
 * Put the tokens into the kernel. The tokens are expected to have
 * been initialized using INIT_AFS_TOKEN().
 *
 * Return 0 on success, -1 on failure.
 */
int
putAFSTokens (AFSTOKENS *tokens)
{
    struct ViceIoctl iob;
    int i;

    for (i = 0; i < tokens->num; i++) {
        iob.in = tokens->tok[i].data;
        iob.in_size = tokens->tok[i].len;
        iob.out = tokens->tok[i].data;
        iob.out_size = tokens->tok[i].len;

        if (pioctl(0, _VICEIOCTL(3), &iob, 0)) {
            fprintf(stderr, "Failed to put token %d into kernel: ", i);
            perror("");
            return (-1);
        }
    }

    return (0);
} /* putAFSTokens */
```

4.4. La routine rtok

Siccome un token ha una durata limitata nel tempo, c'è bisogno di avere a disposizione un tool che ne consenta il rinnovo e che quindi ne prolunghi la validità. Ciò avviene mediante il programma `rtok`. Per poter utilizzare questo kit di rinnovo bisogna conoscere su quale server gira e su quale porta è in ascolto il demone `rktokd`. Inoltre `rtok` non può essere lanciato manualmente da shell ma solamente da un demone LSF come ad esempio `daemon.wrap` oppure `eexec`, altrimenti terminerà istantaneamente la sua esecuzione senza stampare messaggi a video.

Per prima cosa `rtok` controlla che l'utente abbia inserito correttamente password e username. Poi, tramite la routine `sendRenewReq`, effettua la richiesta di rinnovo al demone in ascolto.

4.5. Il server rktokd

Come già accennato prima, `rktokd` è un demone che accetta richieste di rinnovo dei tokens. A differenza del client, il server può essere lanciato da shell usando come parametri `'-p portnumber'` (la porta su cui il demone accetta le richieste) e un file `client_hostfile` contenente la lista degli host autorizzati al rinnovo del token.

4.6. Una visione d'insieme

Nel kit LSF sono presenti anche due script `esub` e `eexec` le cui finalità sono state trattate nei precedenti paragrafi. Tali meccanismi non sono meccanismi dedicati ad AFS ma, più in generale, servono per controllare l'esecuzione dei job e possono essere impiegati congiuntamente per effettuare il trasferimento del token AFS all'host di sottomissione.

Gli script presenti nel *package* potrebbero essere ulteriormente modificati e personalizzati ma, per i nostri scopi, ne analizzeremo solo le parti logicamente più importanti. Cominciamo con lo script `esub`.

```
#!/bin/sh
#
# esub script
# Sample esub script to show how AFS token processing can be supported
# using esub/eexec.
#
# If PGP is used, make sure you set LSF_PGP_NAME to the name you want
# (see README in the LSF AFS tarfile).
LSF_PGP_NAME="lsf"
# Set the directory containing pgp executable if pgp is used
PGP_EXEC_PATH="/usr/local/bin"
```

```
# Comment out the following if pgp is used; i.e., use user's PGPPATH
PGPPATH=""

# Nothing below should be touched.

PATH=${PATH}:/usr/bin:/bin:/usr/local/bin:/local/bin:/sbin:/usr/sbin:/usr/ucb:/usr/sbin:/usr/bsd:${PGP_EXEC_PATH}

# For security reasons, fill in the full path name of EXECDIR.
EXECDIR=
export EXECDIR
if [ ! -d "$EXECDIR" ]
then
    exec 1>&2
    echo "Cannot find EXECDIR. Exiting..."
    exit $LSB_ABORT_VALUE
fi

if [ "$PGPPATH" != "" ] ; then
    USERID=`pgp -kv 2>/dev/null | awk '{print $4}' | tail +4r 2>/dev/null | tail +2
2>/dev/null | xargs echo `
fi

if [ "$PGPPATH" != "" -a -d "$PGPPATH" -a "$USERID" != "" ] ; then
    for userid in $USERID
    do
        if [ "$userid" = "$LSF_PGP_NAME" ] ; then
            $EXECDIR/gettok 2>/dev/null | pgp -ef $LSF_PGP_NAME
            if [ $? -ne 0 ] ; then
                exec 1>&2
                echo "Error running $EXECDIR/gettok. Exiting..."
                exit $LSB_ABORT_VALUE
            fi
            break
        else
            continue
        fi
    done
else
    $EXECDIR/gettok 2>/dev/null
    if [ $? -ne 0 ] ; then
        exec 1>&2
        echo "Error running $EXECDIR/gettok. Exiting..."
        exit $LSB_ABORT_VALUE
    fi
fi
exit $?
```

La parte di nostro interesse è quella riportata su sfondo grigio nella quale è eseguito il programma `gettok` (lo `stderr` del comando è rediretto su `/dev/null`, ed in caso di errore dopo aver rediretto lo `stdout` su `stderr` viene visualizzato il messaggio Error running). Come sappiamo il programma `gettok` stampa sullo *standard output* il token AFS

Tale token, tramite LSF che agisce da *pipe*, costituirà *lo standard input* di `eexec` ossia dello script

```
#!/bin/sh
#
# eexec script
# Sample eexec script to show how AFS token processing can be supported
# using esub/eexec.
#
# If PGP is used, be sure to set ROOT_PGPPATH to the path you want.
# Modify ROOT_PGPPATH for use in your system (see README in the afs tarfile)
ROOT_PGPPATH=/local/etc/.pgp
```

```

# Set the directory containing pgp executable if pgp is used
PGP_EXEC_PATH="/usr/local/bin"
# Comment out the following line if pgp is used; i.e., use user's PGPPATH
PGPPATH=""

RTOK_PORT=""
# Set the rtokd daemons port if token renewal is used
# RTOK_PORT=3333

# Set the AFS server host running the rtokd daemons
# RTOKD_HOST="unixhub"

# Set this to /dev/null if you don't want to log messages
ERRLOG="/dev/null"

# Nothing below should be touched

PATH=${PATH}:/usr/bin:/bin:/usr/local/bin:/local/bin:/sbin:/usr/sbin:/usr/ucb:/usr/sbin:/usr/bsd:${PGP_EXEC_PATH}

# Assume rtok, puttok are in the directory as this executable
# For security reasons, enter the full path name of EXECDIR
EXECDIR=""
export EXECDIR
RTOK="$EXECDIR/rtok -p $RTOK_PORT -s $RTOKD_HOST"

if [ "$PGPPATH" != "" ] ; then
    # User has PGPPATH; assume user is using pgp
    PGPPASSFD=0
    export PGPPASSFD
    # Use root's pgppath to decrypt
    PGPPATH=$ROOT_PGPPATH
    export PGPPATH
    (cat $PGPPATH/.p; cat) | pgp -f 2>>$ERRLOG | $EXECDIR/puttok 2>>$ERRLOG
else
    $EXECDIR/puttok 2>>$ERRLOG
fi

#
# -r means eexec is called recursively from $RTOK
#
if [ "$1" != "-r" -a "$RTOK_PORT" != "" ] ; then
    endtime=`$EXECDIR/gettok -t 2>>$ERRLOG`
    if [ "$endtime" -lt 3600 ] ; then
        $RTOK 2>>$ERRLOG
    fi

    if [ "$LS_EXEC_T" = "START" ] ; then
        # Fork grandchild to stay around for the duration of the job to
        # renew token

        (
            (timeCount=0
            while : ;
            do
                kill -0 $LS_JOBPID >/dev/null 2>&1
                if [ "$?" = "0" ] ; then
                    timeCount=`expr $timeCount + 1`
                    if [ $timeCount -ge 120 ] ; then
                        # Test tokens' end time each hour.
                        endtime=`$EXECDIR/gettok -t 2>>$ERRLOG`
                        timeCount=0
                        if [ "$endtime" -lt 3600 ] ; then
                            $RTOK 2>>$ERRLOG
                        fi
                    else
                        sleep 30
                    fi
                fi
            else
                fi
            fi
        )
    fi

```

```
                exit 0
            fi
        done)&
    )&
    wait
fi
fi
exit $?

```

La parte di nostro interesse è quella riportata su sfondo grigio nella quale viene eseguito il programma puttok (lo stderr del comando è rediretto su \$ERRLOG).

Il programma puttok prende il token dallo *standard input* e lo scrive nel kernel. In tal caso, poiché come detto in precedenza LSF agisce da pipe fra *esub* ed *eexec*, lo *stdin* di *eexec* sarà lo *stdout* di *esub* e quindi il token estratto da *bsub* sarà messo nel kernel da *eexec* con la differenza che quello estratto da *esub* sarà sulla macchina che sottomette mentre *eexec* agirà sull'host di esecuzione.

Lo script *eexec* riportato effettua anche altre operazioni fra cui quella del rinnovo del token.

5. OpenAFS per Windows

Per poter procedere all'analisi del comportamento di AFS in ambiente Windows si è proceduto con l'installazione del package OpenAFS disponibile sul sito <http://www.openafs.org> anche in forma sorgente. A partire da tale kit risulta possibile installare OpenAFS anche in ambiente Windows.

L'operazione di installazione di OpenAFS in ambiente Windows, a partire dai sorgenti, non è risultata particolarmente agevole anche a causa del livello della documentazione disponibile, spesso rilevatasi imprecisa o perlomeno approssimativa. In ogni caso la documentazione dell'attività di installazione di AFS è riportata in Appendice C, una sorta di *quick guide* per l'installazione del prodotto.

Il § 5.2 riporta le routine che sono state esaminate per studiare le problematiche che in ambiente Windows devono essere affrontate per poter leggere il token mentre il successivo 5.3 è relativo alle routine esaminate con la finalità di scrivere il token. La descrizione delle routine è preceduta dal § 5.1 che riporta le strutture dati ed i *function prototypes*.

Nel complesso l'attività di studio descritta in questo paragrafo porrà le basi per la scrittura, in versione puramente prototipale, di due routine, una per la lettura e l'altra per la scrittura del token.

5.1. Strutture dati e function prototypes

Esaminiamo le principali strutture dati utilizzate nelle routine che saranno esaminate nel seguito.

La prima, `krc_principal`, rappresenta in AFS un *principal* usato nel Kernel Token Cache (KTC).

Ricordiamo che tutte le entità sulla rete (sia utenti che servizi) sono riferiti come *principal* mentre il KTC è parte del *Cache Manager*.

```
struct krc_principal {
    char name[MAXKTCNAMELEN];
    char instance[MAXKTCNAMELEN];
    char cell[MAXKTCREALMLEN];
#ifdef AFS_NT40_ENV
    char smbname[MAXSMBNAMELEN];
#endif
};
```

definita in `rxkad.p.h`

Altra struttura utilizzata è `struct krc_token` definita in `auth.h` generata da `auth.p.h`

```
struct krc_token {
    afs_int32 startTime;
```

```

    afs_int32 endTime;
    struct ktc_encryptionKey sessionKey;
    short kvno;
    int ticketLen;
    char ticket[MAXKTCTICKETLEN];
};

```

In questa struttura sono memorizzati tutti i campi del Token:

- *startTime* ed *endTime* rappresentano rispettivamente il momento in cui il token viene assegnato dal server e il momento in cui non sarà più utilizzabile. È da notare che la routine *ka_UserAuthenticateGeneral* (file *src\Kauth\user_nt.c*), utilizzata da Klog, va a settare a 0 lo *startTime* a differenza della stessa routine del kit LSF che lo setta al corretto istante in cui il token è stato assegnato;
- *sessionKey* è spiegata nel dettaglio successivamente;
- *kvno*: Key Version Number associato al Session encryption key;
- *ticketLen* rappresenta la lunghezza del ticket (secret token);
- *ticket* contiene il secret token.

La struttura *ktc_encryptionKey* ha la definizione:

```

struct ktc_encryptionKey {
    char data[8];
};

```

e quindi corrisponde a quello che nei paragrafi precedenti abbiamo chiamato l'*HandShakeKey* del *ClearToken*.

L'ultima delle strutture utilizzate è la *ClearToken* in cui sono presenti i campi relativi al clear token:

```

struct ClearToken {
    afs_int32 AuthHandle;
    char HandShakeKey[8];
    afs_int32 ViceId;
    afs_int32 BeginTimestamp;
    afs_int32 EndTimestamp;
};

```

Questa struttura è presente anche nel codice del kit LSF

- *AuthHandle* rappresenta il kvno;
- *ViceId* viene settato a 37 con una nota dei programmatori;
- *BeginTimestamp* rappresenta lo *StartTime* del token;
- *EndTimestamp* rappresenta l'*EndTime* del token;
- *HandShakeKey* rappresenta la chiave di sessione.

Le dichiarazioni delle funzioni che saranno analizzate nei seguenti paragrafi sono:

```

int ktc_ListTokens(int cellNum, int *cellNumP, struct ktc_principal *server)

int ktc_GetToken(struct ktc_principal *server, struct ktc_token *token,
    int tokenLen, struct ktc_principal *client)

int ktc_SetToken(struct ktc_principal *server, struct ktc_token *token,
    struct ktc_principal *client, int flags)

```

5.2. La lettura del token

Per esaminare le modalità di lettura in ambiente Windows si è partiti dal sorgente relativo al comando `tokens` di AFS che ha, come output, l'elenco dei token presenti sul sistema.

Il file è presente in `openafs-1.x.x\src\WINNT\afsd\ctokens.c`. All'interno del `main` di tale file sono poi presenti le invocazioni di altre routine e precisamente la `ktc_ListTokens` e la `ktc_GetToken` ambedue definite in `openafs-1.x.x\src\auth\ktc_nt.c`.

La `ktc_ListTokens` preleva la lista dei token disponibili sul sistema e, nel caso ci siano dei token, essi vengono letti completamente mediante la routine `ktc_GetToken`.

5.2.1. La routine `ktc_ListTokens`

La routine è definita nel file `openafs-1.x.x\src\auth\ktc_nt.c` ed ha primariamente lo scopo di restituire il *principal* del *server* e di conseguenza è anche impiegata per accertarsi della presenza di token.

Per restituire il *principal* del *server* deve comunque leggere il token al fine di impostare `server->cell` (al valore enea.it).

La *Figura 4 Strutture dati coinvolte nella routine `ktc_ListTokens`* mette in evidenza le principali strutture coinvolte nella routine, mentre di seguito si riporta il codice con dei commenti per agevolarne la comprensione.

```
int ktc_ListTokens(int cellNum, int *cellNumP, struct ktc_principal *server)
{
    struct ViceIoctl iob;
    char tbuffer[TBUFFERSIZE];
    char *tp, *cp;
    int newIter, ticketLen, temp;
    int code;
    HANDLE ktcMutex = NULL;

#ifdef AFS_WIN95_ENV
    ktcMutex = CreateMutex(NULL, TRUE, AFSGlobalKTCMutexName);
    if (ktcMutex == NULL)
        return KTC_PIOCTLFAIL;
    if (GetLastError() == ERROR_ALREADY_EXISTS) {
        if (WaitForSingleObject(ktcMutex, INFINITE) != WAIT_OBJECT_0) {
            CloseHandle(ktcMutex);
            return KTC_PIOCTLFAIL;
        }
    }
#endif /* AFS_WIN95_ENV */

    tp = tbuffer; /* tbuffer è un array di caratteri che viene
                  * scorso tramite l'iteratore tp */
}
```

```

memcpy(tp, &cellNum, sizeof(cellNum)); /* Nei primi 4 byte di tbuffer
                                         * metto il CellName */
tp += sizeof(cellNum); /* Avanzo il puntatore di 4 */

/* Costruzione struttura ViceIoctl da passare alla funzione pioctl */
iob.in = tbuffer; /* iob.in punta a tbuffer */
iob.in_size = (long)(tp - tbuffer); /* iob.in_size = 4 */
iob.out = tbuffer; /* iob.out punta a tbuffer */
iob.out_size = sizeof(tbuffer);

code = pioctl(0, VIOCGETTOK, &iob, 0); /* in tbuffer verrà scritto l'output
                                         * della pioctl che conterrà tutte le
                                         * informazioni sul token */

#ifdef AFS_WIN95_ENV
ReleaseMutex(ktcMutex);
CloseHandle(ktcMutex);
#endif /* AFS_WIN95_ENV */

if (code) {
    if (code == -1) {
        if (errno == ESRCH)
            return KTC_NOCELL;
        else if (errno == ENODEV)
            return KTC_NOCM;
        else if (errno == EINVAL)
            return KTC_INVALID;
        else if (errno == EDOM)
            return KTC_NOENT;
        else
            return KTC_PIOCTLFAIL;
    } else
        return KTC_PIOCTLFAIL;
}
cp = tbuffer;

/* new iterator */
memcpy(&newIter, cp, sizeof(newIter)); /* Copio i primi 4 byte di tbuffer
                                         * in newIter (campo vuoto) */
cp += sizeof(newIter); /* Mi sposto in avanti di 4 su tbuffer */

/* ticket length */
memcpy(&ticketLen, cp, sizeof(ticketLen)); /* Copio in ticketLen la
                                             * lunghezza del ticket */
cp += sizeof(ticketLen);

/* skip over ticket */
cp += ticketLen; /* Salto il ticket */

/* clear token size */
memcpy(&temp, cp, sizeof(temp)); /* Copio in temp la lunghezza
                                  * del Clear Token */
cp += sizeof(temp); /* Mi sposto in avanti di 4 su tbuffer */
if (temp != sizeof(struct ClearToken))
    return KTC_ERROR;

/* skip over clear token */
cp += sizeof(struct ClearToken); /* Salto il Clear Token */

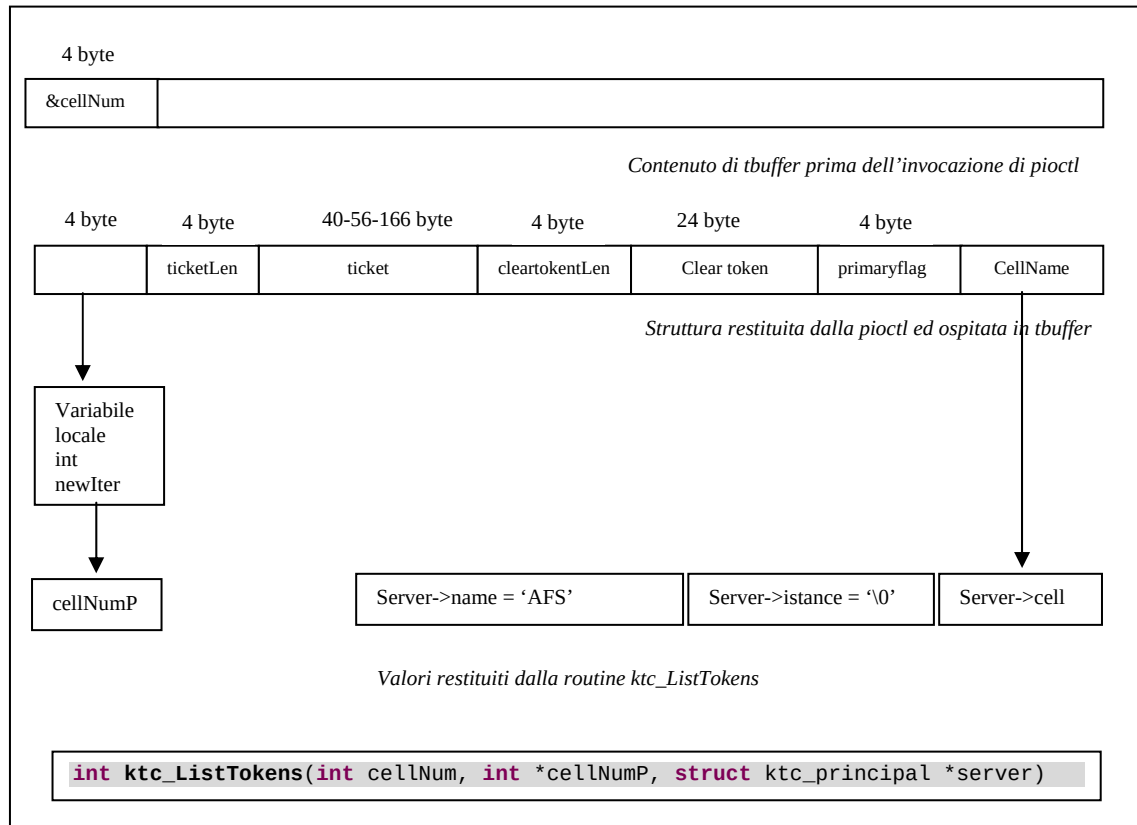
/* skip over primary flag */
cp += sizeof(temp); /* Salto il primary flag */

/* cell name is here */
/* set return values */
strcpy(server->cell, cp); /* server.cell impostato a nome cella */
server->instance[0] = '\0'; /* server.instance settato a Null */
strcpy(server->name, "afs"); /* server.name settato a "afs" */

*cellNumP = newIter; /* cellNumP = newIter (primi 4 byte di tbuffer) */
return 0;

```

}

Figura 4 Strutture dati coinvolte nella routine `ktc_ListTokens`

5.2.2. La routine ktc_GetToken

La routine ktc_GetToken, definita nel file ktc_nt.c, preleva il token dal kernel e lo memorizza in una struttura di tipo ktc_token.

In maniera analoga a quanto eseguito dalla getAFSToken del kit LSF, la funzione si basa sull'invocazione della *system call* pioctl effettuata dopo l'impostazione dei campi della struttura ViceIoctl, parametro della pioctl. La *system call* restituisce il suo output tramite il campo iob.out (o equivalentemente tbuffer).

La routine pioctl è definita in pioctl_nt.c (memorizzato in openafs\src\sys).

È importante evidenziare che, quando viene scritto un token in memoria, la *session key* non viene settata come tutti gli altri campi del token, bensì inviata via RPC. Per questo motivo nella seguente routine si effettua una chiamata alla funzione `receive_key(uuid, token->sessionKey.data)` per leggere e decriptare appunto la *session key*.

La routine provvede quindi ad impostare i vari campi della struttura ktc_token prevista in output sulla base di quanto restituito dalla *system call*.

Le strutture utilizzate sono descritte nella *Figura 5 Strutture dati coinvolte nella routine ktc_GetTokens* e di seguito si riporta il codice opportunamente commentato.

```
/* ktc_GetToken */
```

```
Int ktc_GetToken(struct ktc_principal *server, struct ktc_token *token,
                 int tokenLen, struct ktc_principal *client)
{
    struct ViceIoctl8 iob;
    char tbuffer[TBUFFERSIZE];
    char *tp, *cp;
    char *ticketP;
    int ticketLen, temp;
    struct ClearToken ct;
    char *cellName;
    int cellNameSize;
    int maxLen;
    int code;
    RPC_STATUS status;
    afs_uuid_t uuid;
    HANDLE ktcMutex = NULL;

    tp = tbuffer;

    /* check to see if the user is requesting tokens for a principal
     * other than afs. If so, check the local token cache.
     */

```

```
8 /* looks like pioctl block from the Unix world */
   typedef struct ViceIoctl {
       long in_size;
       long out_size;
       void *in;
       void *out;
   } viceIoctl_t;
```

```

    if (strcmp(server->name, "afs")) {
        return GetLocalToken(server, token, tokenLen, client);
    }

    /* cell name */
    strcpy(tp, server->cell);
    tp += strlen(server->cell) + 1;

    /* uuid */
    status = UuidCreate((UUID *) & uuid);
    memcpy(tp, &uuid, sizeof(uuid));
    tp += sizeof(uuid);

    /* Settaggio struttura iob da passare alla funzione pioctl */
    iob.in = tbuffer;                /* iob.in punta a tbuffer */
    iob.in_size = (long)(tp - tbuffer); /* iob.in_size = 4 */
    iob.out = tbuffer;               /* iob.out punta a tbuffer */
    iob.out_size = sizeof(tbuffer); /* iob.out_size dimensione di tbuffer */

#ifdef AFS_WIN95_ENV
    ktcMutex = CreateMutex(NULL, TRUE, AFSGlobalKTCMutexName);
    if (ktcMutex == NULL)
        return KTC_PIOCTLFAIL;
    if (GetLastError() == ERROR_ALREADY_EXISTS) {
        if (WaitForSingleObject(ktcMutex, INFINITE) != WAIT_OBJECT_0) {
            CloseHandle(ktcMutex);
            return KTC_PIOCTLFAIL;
        }
    }
#endif /* AFS_WIN95_ENV */

    /* Riempie tbuffer con tutte le informazioni sul token */
    code = pioctl(0, VIOCNEWGETTOK, &iob, 0);
    if (code) {
#ifdef AFS_WIN95_ENV
        ReleaseMutex(ktcMutex);
        CloseHandle(ktcMutex);
#endif /* AFS_WIN95_ENV */
        if (code == -1) {
            if (errno == ESRCH)
                return KTC_NOCELL;
            else if (errno == ENODEV)
                return KTC_NOCM;
            else if (errno == EINVAL)
                return KTC_INVAL;
            else if (errno == EDOM)
                return KTC_NOENT;
            else
                return KTC_PIOCTLFAIL;
        } else
            return KTC_PIOCTLFAIL;
    }
#ifdef AFS_WIN95_ENV
    /* get rid of RPC for win95 build */
    /* RPC to receive session key */
    status = receive_key(uuid, token->sessionKey.data);

    ReleaseMutex(ktcMutex);
    CloseHandle(ktcMutex);

    if (status != RPC_S_OK) {
        if (status == 1)
            strcpy(rpcErr, "RPC failure in AFS gateway");
        else
            DceErrorInqText(status, rpcErr);
        if (status == RPC_S_SERVER_UNAVAILABLE
            || status == EPT_S_NOT_REGISTERED)
            return KTC_NOCMRPC;
        else
            return KTC_RPC;
    }

```

```

    }
#endif /* AFS_WIN95_ENV */

    cp = tbuffer;

    memcpy(&ticketLen, cp, sizeof(ticketLen)); /* Copio i primi 4 byte di tbuffer
                                                * che determinano la lunghezza
                                                * del ticket */
    cp += sizeof(ticketLen); /* Sposto in avanti il puntatore di 4 */

    ticketP = cp; /* Memorizzo la posizione del ticket */
    cp += ticketLen; /* Sposto il puntatore dopo il ticket */

    /* size of clear token */
    memcpy(&temp, cp, sizeof(temp)); /* Copio in temp la dimensione
                                      * di Clear Token */
    cp += sizeof(temp); /* Sposto in avanti il puntatore
                        * della dimensione di Clear Token */

    if (temp != sizeof(ct))
        return KTC_ERROR;

    /* clear token */
    memcpy(&ct, cp, temp); /* Copio in ct il Clear Token */

    cp += temp; /* Sposto il puntatore alla fine del Clear Token */

    cp += sizeof(temp); /* Salt oil Primary flag */

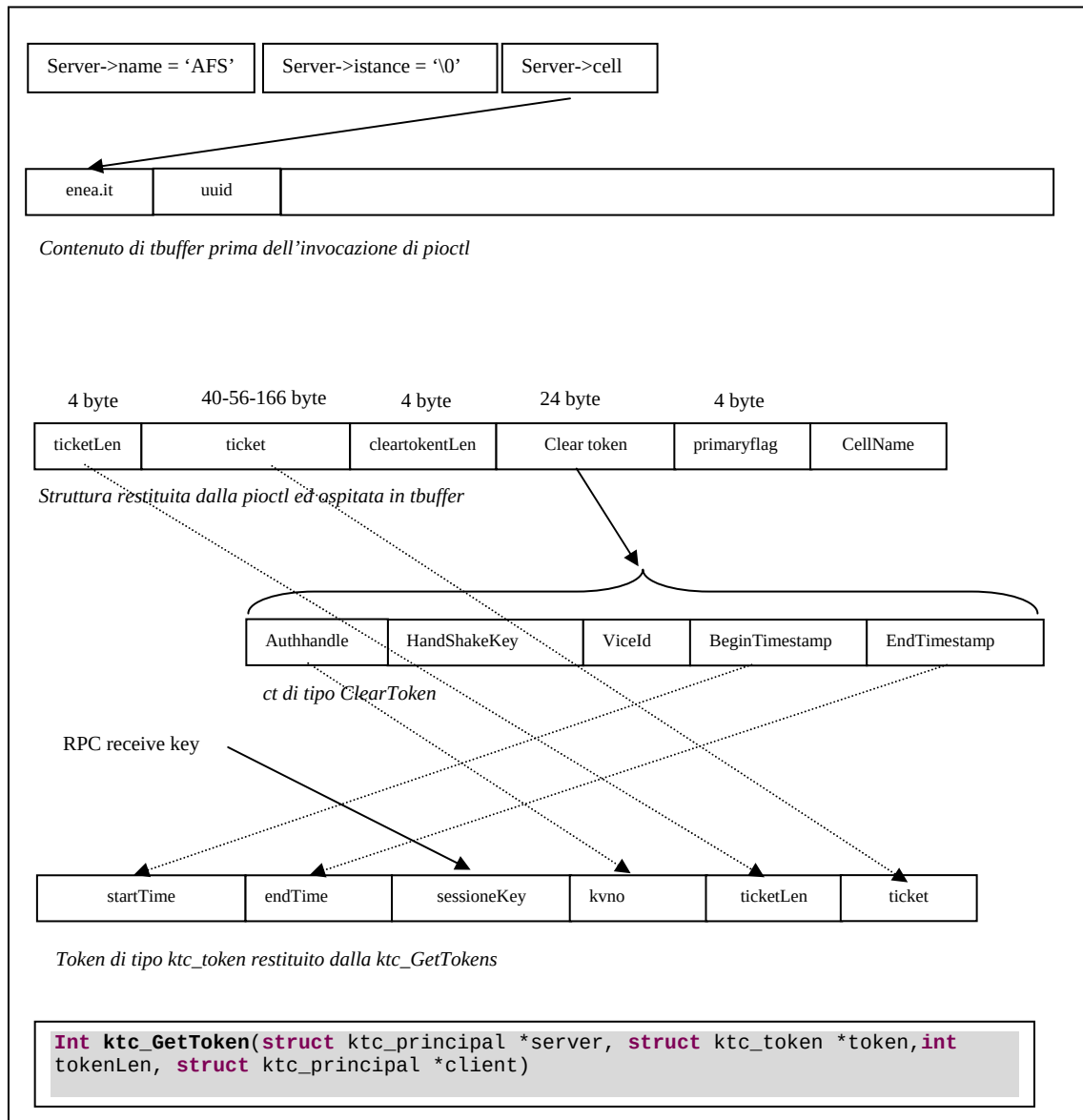
    cellName = cp; /* memorizzo il cellName */
    cellNameSize = (int)strlen(cp);
    cp += cellNameSize + 1; /* Sposto il puntatore dopo cellName */

    /* user name is here */

    /* check that ticket will fit
     * this compares the size of the ktc_token allocated by the app
     * which might be smaller than the current definition of MAXKTCTICKETLEN
     */
    maxLen = tokenLen - sizeof(struct ktc_token) + MAXKTCTICKETLEN;
    if (maxLen < ticketLen)
        return KTC_TOOBIG;

    /* Setto i campi della struttura Clear Token */
    memcpy(token->ticket, ticketP, ticketLen);
    token->startTime = ct.BeginTimestamp;
    token->endTime = ct.EndTimestamp;
    if (ct.AuthHandle == -1)
        ct.AuthHandle = 999;
    token->kvno = ct.AuthHandle;
#endif AFS_WIN95_ENV
    /*
     * Session key has already been set via RPC
     */
#else
    memcpy(&token->sessionKey, ct.HandShakeKey, sizeof(ct.HandShakeKey));
#endif /* AFS_WIN95_ENV */
    token->ticketLen = ticketLen;
    if (client) {
        strcpy(client->name, cp);
        client->instance[0] = '\0';
        strcpy(client->cell, cellName);
    }
    return 0;
}

```

Figura 5 Strutture dati coinvolte nella routine `ktc_GetTokens`

5.3. La scrittura del token

Per esaminare le modalità di scrittura in ambiente Windows si è partiti dal sorgente relativo al comando `klog` di AFS che serve per ottenere e rinnovare un token. Tale file è presente in `openafs-1.x.x\src\aklog\aklog.c`.

Il comando `klog` provvede ad una serie di operazioni più o meno complesse in quanto è preposto a contattare l'*Authentication Server* sia per l'autenticazione che per l'ottenimento del token ma per i fini dell'attività ci si è soffermati sulla fase in cui, ottenuto il token, lo stesso deve essere scritto nel sistema. Tale operazione è svolta dalla routine `ktc_SetToken`.

5.3.1. La routine `ktc_SetToken`

La routine `ktc_SetToken`, definita nel file `ktc_nt.c`, provvede a settare il token nel kernel del sistema operativo ed è quindi presa come base di partenza per la costruzione della routine `puttok` per windows.

Prende in input tre strutture, di cui una, di tipo `ktc_token`, che contiene i principali campi del token che poi sarà scritto nel kernel. Le altre due sono di tipo `ktc_principal` e contengono informazioni sul *client* e sul *server* coinvolti in questa operazione.

In maniera analoga a quanto eseguito dalla `putAFSToken` del kit LSF, la funzione si basa sull'invocazione della *system call* `pioctl` previa impostazione del campo `iob.in` o equivalentemente `tbuffer`.

Simmetricamente a come avviene nella routine `ktc_GetToken`, in primo momento, invece di inviare in chiaro la *session key*, questa routine la setta a 0 per poi mandarla in un secondo momento, criptata, via RPC. Il tutto avviene tramite la funzione `send_key(uuid, token->sessionKey.data)`.

Il seguente codice è commentato per agevolarne la comprensione⁹.

```
Int ktc_SetToken(struct ktc_principal *server, struct ktc_token *token,
                struct ktc_principal *client, int flags)
{
    struct ViceIoctl iob;
    char tbuffer[TBUFFERSIZE];
    char *tp;
    struct ClearToken ct;
    int temp, i;
    int code;
    RPC_STATUS status;
    afs_uid_t uuid;
```

⁹ In questo file, ma anche in altri che sono stati esaminati durante le attività di studio del kit OpenAFS, è presente un commento del tipo `/* XXX */` che sembra essere utilizzato dai programmatori di OpenAFS per evidenziare aspetti non ancora definitivamente chiariti. Nel caso specifico, il commento XXX accompagna la linea di codice in cui si imposta a 37 il campo `ViceID`.

```

HANDLE ktcMutex = NULL;
if (token->ticketLen < MINKCTICKETLEN || token->ticketLen > MAXKCTICKETLEN)
{
    return KTC_INVALID;
}
printf("\nserver.name: %s", server->name);
if (strcmp(server->name, "afs")) {
    return SetLocalToken(server, token, client, flags);
}
tp = tbuffer; /* iteratore su tbuffer */

memcpy(tp, &token->ticketLen, sizeof(token->ticketLen)); /* copio lunghezza del
* ticket in tbuffer */
tp += sizeof(token->ticketLen); /* Mi sposto di 4 in avanti su tbuffer */

memcpy(tp, token->ticket, token->ticketLen); /* Copio il ticket su tbuffer */
tp += token->ticketLen; /* Sposto il puntatore alla fine del ticket */

ct.AuthHandle = token->kvno; /* Copio kvno nel Clear token */
/*
 * Instead of sending the session key in the clear, we zero it,
 * and send it later, via RPC, encrypted.
 */
#ifdef AFS_WIN95_ENV
memset(ct.HandShakeKey, 0, sizeof(ct.HandShakeKey));
#else
memcpy(ct.HandShakeKey, &token->sessionKey, sizeof(token->sessionKey));
#endif

ct.BeginTimestamp = token->startTime;
ct.EndTimestamp = token->endTime;
if (ct.BeginTimestamp == 0)
    ct.BeginTimestamp = 1;

/* We don't know from Vice ID's yet */
ct.ViceId = 37; /* XXX */
if (((ct.EndTimestamp - ct.BeginTimestamp) & 1) == 1)
    ct.BeginTimestamp++; /* force lifetime to be even */

temp = sizeof(struct ClearToken);
memcpy(tp, &temp, sizeof(temp)); /* Copio in tbuffer la dimensione della
* struttura Clear Token */
tp += sizeof(temp); /* Mi sposto di 4 in avanti su tbuffer */
memcpy(tp, &ct, sizeof(ct)); /* Copio in tbuffer il Clear Token */
tp += sizeof(ct); /* Mi sposto alla fine del Clear Token */

/* flags; on NT there is no setpag flag, but there is an
 * integrated logon flag */
temp = ((flags & AFS_SETTOK_LOGON) ? PIOCTL_LOGON : 0); /* Primary flag */

memcpy(tp, &temp, sizeof(temp)); /* Copio il Primary flag in tbuffer */
tp += sizeof(temp); /* Mi sposto dopo il primary flag */

temp = (int)strlen(server->cell);
if (temp >= MAXKTCREALMLEN)
{
    return KTC_INVALID;
}
strcpy(tp, server->cell); /* Copio server cell in tbuffer */
tp += temp + 1; /* Mi sposto alla fine fine di server cell */

temp = (int)strlen(client->name); /* Copio in tbuffer il client name */
if (temp >= MAXKTCNAMELEN) {
    return KTC_INVALID;
}
strcpy(tp, client->name);
tp += temp + 1;

/* we need the SMB user name to associate the tokens with in the

```

```

    * integrated logon case. */

    if (flags & AFS_SETTOK_LOGON) {
        if (client->smbname == NULL) {
            temp = 0;
        }
        else
            temp = (int)strlen(client->smbname);
        if (temp == 0 || temp >= MAXKTCNAMELEN) {
            return KTC_INVALID;
        }
        strcpy(tp, client->smbname); /* Copio il Samba Name */
        tp += temp + 1;
    }

    /* uuid */
    status = UuidCreate((UUID *) & uuid);
    memcpy(tp, &uuid, sizeof(uuid)); /* Copio l'uuid in tbuffer */
    tp += sizeof(uuid);

#ifdef AFS_WIN95_ENV
    ktcMutex = CreateMutex(NULL, TRUE, AFSGlobalKTCMutexName);
    if (ktcMutex == NULL)
        return KTC_IOCTLFAIL;
    if (GetLastError() == ERROR_ALREADY_EXISTS) {
        if (WaitForSingleObject(ktcMutex, INFINITE) != WAIT_OBJECT_0) {
            CloseHandle(ktcMutex);
            return KTC_IOCTLFAIL;
        }
    }

    /* RPC to send session key */
    status = send_key(uuid, token->sessionKey.data);
    if (status != RPC_S_OK) {
        if (status == 1)
            strcpy(rpcErr, "RPC failure in AFS gateway");
        else
            DceErrorInqText(status, rpcErr);
        if (status == RPC_S_SERVER_UNAVAILABLE ||
            status == EPT_S_NOT_REGISTERED) {
            ReleaseMutex(ktcMutex);
            CloseHandle(ktcMutex);
            return KTC_NOCMRPC;
        }
        else {
            ReleaseMutex(ktcMutex);
            CloseHandle(ktcMutex);
            return KTC_RPC;
        }
    }
}
#endif /* AFS_WIN95_ENV */

/* set up for ioctl */
iob.in = tbuffer;
iob.in_size = (long)(tp - tbuffer);
iob.out = tbuffer;
iob.out_size = sizeof(tbuffer);

code = ioctl(0, VIOCSETTOK, &iob, 0);

#ifdef AFS_WIN95_ENV
    ReleaseMutex(ktcMutex);
    CloseHandle(ktcMutex);
#endif /* AFS_WIN95_ENV */

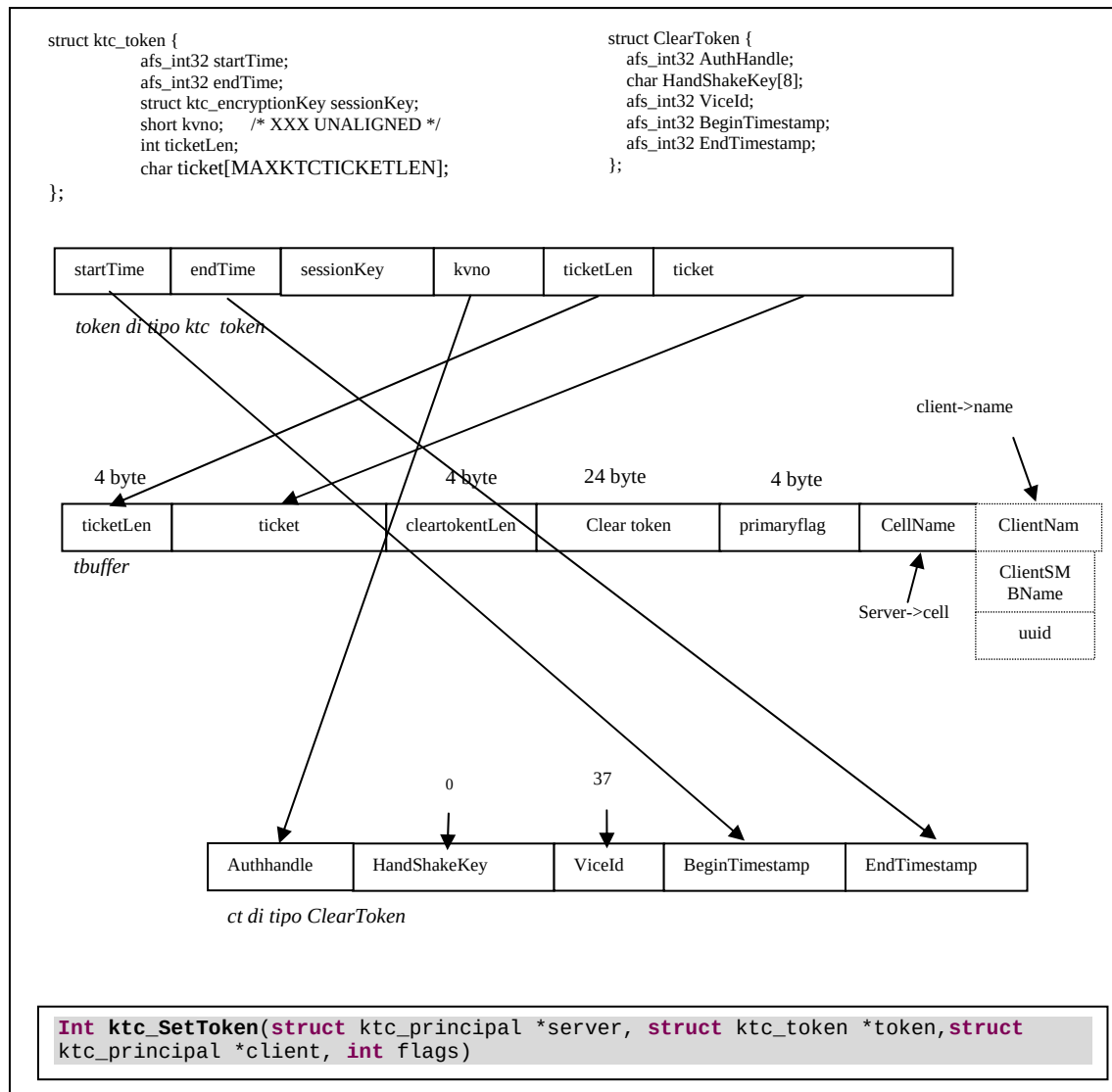
    if (code) {
        if (code == -1) {
            if (errno == ESRCH)
                return KTC_NOCELL;
            else if (errno == ENODEV)
                return KTC_NOCM;
        }
    }

```

```

    else if (errno == EINVAL)
        return KTC_INVALID;
    else
        return KTC_PIOCTLFAIL;
} else
    return KTC_PIOCTLFAIL;
}
return 0;
}

```

Figura 6 Strutture dati coinvolte nella routine `ktc_setToken`

Appendice A -Programming with the Andrew Cache Manager¹⁰

¹⁰ The cache manager is your agent in accessing information stored in AFS. When you access a file, the cache manager on your client machine requests the file from the appropriate file server machine and stores, or caches, a copy of it on your client machine's local disk. Application programs on your client machine use the local, cached copy of the file. This improves performance because it is much faster to use a local file than to send requests for file data across the network to the file server. Saving your file sends the changed file back to the appropriate file server where the file is stored. In campus labs, you cannot save files to the workstation's local drive. When you log out, the C: drive or local disk is cleared. However, because files are saved by default to your home directory in AFS, unless you change the path, your files are safely stored on network file servers, which are also backed up nightly. Just remember to save often, which writes your data to permanent storage on the file server, and do not use the local drive for more than temporary storage.

1. Introduction

This manual describes the programmer's interface to the Andrew File System. It is comprised of two sections. Section intro describes standard Unix file system facilities that have been modified in AFS, and so function differently for files stored in the Andrew File System. Section int describes the *pioctl* system call interface to the Andrew File System, which allows a user program to apply special functions to files stored in the Andrew File System. Most of these operations do not make sense when applied to files stored on the regular Unix file system and will return an error code when so applied. Examples of these operations include changing access lists associated with directories and finding the quotas associated with AFS volumes.

2. Semantic Differences

2.1. Intro

This section describes those functions that are different in the Andrew File System than in a standard Unix file system. When appropriate, if a standard programming technique does not work in the Andrew File System, the standard way to achieve the same effect in the Andrew File System is presented. This section unfortunately presents a hodge-podge of differences.

2.2. File Protection and Access Control Lists

File protection is handled quite differently in the Andrew File System from the Unix file system. In the Unix file system, protection of a file is defined by the protection mode bits. For example, to read a file in the Unix file system, you need "x" access to all of the ancestor directories of the file (in order to be able to search them), as well as "r" access to the final file at the end of the pathname. Each directory in the Andrew File System has an associated **access control list**. Access lists are always associated with directories, not files. Files are controlled by the access control list associated with their parent directory. An access control list consists of a list of **access control entries**, each of which lists the access rights the named user or group has to objects in the directory. Two special groups are interesting in particular:

the group *system:anyuser* names a group containing all users of the system, and the group *system:authuser* names a group containing all *authenticated* users of the system (thus excluding the user *anonymous*).

The Andrew File System implements all 12 Unix protection mode bits, but only for compatibility with programs that store extraneous information in these bits. The mode bits have somewhat different meaning than in the regular Unix file system. The *chmod* and *stat* system calls alter and retrieve all 12 bits, as usual. However, for nearly all operations, the low order six protection bits (the "group" and "other" protection bits) are completely ignored. All protection is governed by the "owner" rwx mode bits combined with the rights granted by the access control list associated with the file's parent directory. The access rights granted to a user making a request by an access control list are determined as follows: For every access

control entry naming a group containing the user, or every entry naming the user directly, the rights associated with that entry are added to the rights already granted to the user. Thus a user can obtain read rights by virtue of membership in one group and write rights through another group.

The rights that can be granted are abbreviated by one character:

- (r)ead Read any file in the directory.
- (w)rite Write any existing file in the directory.
- (i)nsert Create new files or subdirectories in the directory.
- (d)eleate Remove files or subdirectories in the directory.
- (l)ookup List the names of the files in the directory.
- (a)dminister Change the access list on that directory. Users are automatically assigned administer rights to their home directories and any directories created in them.
- (k) Allows read or write locks to be placed on any file in the directory using the flock system call.

When accessing a file in Andrew, the access control list is first consulted, resulting in the granting of some subset of the rights `rwldwka` to the user. Next, the owner mode bits are checked. If the "r" bit is not on in the Unix owner protection mode bits, read access is removed from the set of rights granted via the access control list. Similarly, if the owner's "w" bit is not set, the write access is removed from the rights granted. It turns out that there are actually two access lists associated with a directory. The second is called the **negative rights list** and is used as follows: If there are negative rights associated with a directory, the same access list evaluation scheme is applied to it after the rights granted by the normal access list are determined. Any rights listed in the negative rights list are then removed from the final list of access rights granted. For instance, if a negative access control list contains the entry `system:anyuser rldwka`, no one at all will be able to do anything to any of the files in this directory, since everyone matches `system:anyuser`. Negative access rights are quite subtle in their behavior and only experienced users should use them.

Finally, the Andrew File System is still in development. It no doubt contains bugs that manifest themselves as security holes. You should follow the so called "Ellsberg," or "North," rule:
Never keep anything in a computer whose revelation will cause you more embarrassment than the revelation that someone else is reading your files illicitly will cause the reader.
 Or, better yet
Never keep anything personal on a computer.

2.3. Open Modes

In AFS, the Unix mode `O_APPEND` does append data to the end of a file, but does not guarantee that two different writes from different workstations will be serialized properly.

2.4. File Consistency

In the Unix file system, every time a write system call completes, the next read system call directed to that file will read the newly-written data. While this is true in the Andrew File System within a single workstation, when more than one workstation is involved, the rule must be relaxed: Data written to a file can be read by any program after the last writer closes or *fsync*'s the file. That is, the Andrew File System only creates a new version of the file at the file server when the last writer of a file on a workstation closes that file, or when the file is forced through to the server by the issuing an *fsync* system call. In general, then, the data will not necessarily be visible until after the writer closes the file. Since most programs close files when they are done (or when they exit, automatically), this rarely causes any troubles.

2.5. SetUID Programs

The concept of a *setuid* program is somewhat ill-defined in a network environment where workstations are untrusted. In the AFS model, the only user for which the *setuid* bit makes sense is the root user, since the root user does not have special access to files in the Andrew File System. Running a program as *root* confers special privileges only with regard to the workstation kernel. Any file can be marked as *setuid*, but only system administrators can change the owner of a file (via the *chown* system call).

2.6. Locking Files

The biggest difference between locking in Unix and locking in AFS is that in the Andrew File System, the *flock* system call never blocks. Regardless of whether *LOCK_NB* is specified in the *flock* call, if the file is already locked by someone else, the *flock* system call will get an *EWOLDBLOCK* error, and fail immediately.

2.7. Hard Links

There are some restrictions on hard links in the Andrew File System, arising from two causes. First, all linked files must be contained in the same directory, in order that our access control mechanism be well-defined. Thus, one may only create hard links to files contained within the same Andrew File System directory. Second, in order to keep the file system organized as a tree, the Andrew File System does not allow hard links to directories.

2.8. Volumes, Partitions and Quotas

Volumes in the Andrew File System can be thought of as small mountable file systems. Each one appears as a connected subtree within the Andrew File System. All of the files in a particular volume are located on the same server and can be easily moved from server to server. For most users, all of the files in their home directories are located in one volume. Disk quotas are associated with each volume. All volumes have two associated disk quotas: a limiting disk space quota and a guaranteed disk quota. The minimum quota indicates the amount of storage guaranteed by the server to be available at any time for this volume. Other volumes may be prevented from growing in order to provide this guarantee. The minimum quota is expressed in units of 1024 byte blocks, and is commonly set to 0. The maximum quota

specifies the maximum amount of storage, in 1024 byte blocks, that the volume is allowed to contain. A maximum quota of 0 means, paradoxically, that the volume can contain an unlimited amount of storage. If a file store operation would cause a volume to exceed its quota, the store fails with the global variable *errno* set to *EDQUOT*. If a volume is over quota, attempts to create new files also fail with the same error.

2.9. Errors

When system calls fail in the Unix system, they usually return the value -1 and return a more detailed error code in the global variable *errno*. Some of these error codes have different meaning in the Andrew File System than in the Unix file system. These include

ENOSPC	The disk partition on the server containing the relevant file is full. This error will be difficult to fix yourself; call the operator's hotline and report the condition.
EDQUOT	The volume on which the relevant file is to be stored is over its disk quota. Either request more quota or delete some files.
ETIMEDOUT	The file server to which the operation was directed is not responding.

3. Special Functions

While most operations on files stored in the Andrew File System are performed using the standard Unix file system calls, certain operations have no Unix file system equivalents. There are two interfaces to the Andrew File System, one using the standard *ioctl* system call and one using a new system call named *pioctl*, for accessing these functions. The *pioctl* system call takes the same parameters as the *ioctl* system call, except that the first parameter to *pioctl* is the pathname to the file in question rather than a file descriptor.

To use these calls, you must include `/usr/andy/include/afs/venus.h`. The traditional method of including files from such esoteric directories is to put `"-I/usr/andy/include"` as a CFLAG in your make file, telling the C pre-processor to search this directory for include files. Your program should contain these lines:

```
#include <sys/types.h> /* defines useful types for venus.h */
#include <afs/venus.h>
```

at its head.

The *include* file defines a structure called a "struct ViceIoctl", used as a parameter to all of the Andrew File System *ioctl* and *pioctl* system calls. It defines two pointers, *in* and *out*, and two short integers, *in_size* and *out_size*. Data is passed to the Andrew File System in the input buffer (pointed to by the *in* component and containing *in_size* bytes) and results are retrieved from the output buffer (pointed to by the *out* component, and containing *out_size* bytes). Either buffer may be omitted by specifying a size of 0. If the buffer is too small for the resulting output parameter, the call will fail and the global variable *errno* will be set to *EINVAL*.

The *include* file also defines a number of constants, such as *VIOCGETAL*, that specify the operation that the *pioctl* system call is to perform. These are defined in terms of another macro, *_VICEIOCTL*, which should no longer be used in your programs directly. This

document uses the new-style names for the operations exclusively; if you have to convert a old-style name to a new-style name, consult `usr/andy/include/afs/venus.h` for the mapping.

When using the `pioctl` call, you must link in the subroutine providing a C language interface to the `pioctl` system call, too. This is accomplished by linking in `/usr/andrew/server/lib/libsys.a` when you load your program (generally your "cc -o ..." line in your makefile). Failure to do so will result in the loader (ld) complaining about an undefined symbol named `_pioctl` (as well as `_.pioctl` on the RT/PC). Generally, you simply add `"/usr/andrew/server/lib/syslib.a"` to the end of the cc line that generates the final object file.

Here are the Andrew File System specific `ioctl`s and `pioctl`s:

3.1. **IOCTLs**

Here is an example of how to call an `ioctl` function. This particular function directs AFS to delay the return of a `close` system call until the file being written has been safely stored back to its file server

```
struct ViceIoctl blob; /* declare a ViceIoctl for AFS ioctl
calls */
blob.in_size = 0;      /* no input parameters */
blob.out_size = 0;     /* no output parameters */
returnCode = ioctl(fd, VIOCCLOSEWAIT, &blob);
if (returnCode < 0) perror("ioctl");
```

3.1.1. **VIOCCLOSEWAIT**

Wait for close. No input parameters and no output parameters. This call only makes sense on file descriptors open in a writable mode. The close system call to be performed on this file descriptor should wait until the file has been safely transferred before returning. Under normal circumstances, the close system call for a file that has been opened in write mode returns as soon as all of the callbacks have been broken on a file, ensuring correct synchronization among multiple workstations, but allows the actual data transfer to proceed independently as a background process.

3.1.2. **VIOCBABORT**

This call is obsolete, and is no longer supported.

3.1.3. **VIOCIGETCELL**

Get cell name associated with open file descriptor. If the file is not in the Andrew file system, this call returns an error (ENOTTY). Otherwise, it return one output parameter: the null-terminated name of the cell containing the file open on this file descriptor. The use of this call is the recommended way of telling whether or not a file is stored in the Andrew file system.

3.2. **PIOCTLs**

The `pioctl` system call is very similar to the `ioctl` system call, with two exceptions. First, the relevant file is passed via a pathname as the first parameter. Second, there is a fourth parameter, which determines what file should be used if the file specified is a symbolic link. If this parameter is 1, then the symbolic link is followed to its destination and the `pioctl` is applied to that destination file. If the parameter is 0, however, the `pioctl` is applied to the symbolic link itself. Note that this fourth parameter only applies to a symbolic link if it is the last component of the pathname; symbolic links encountered in intermediate components are

always followed. In the following, the **pathname parameter** refers to the first parameter to the call. The **follow parameter** refers to the fourth parameter to the *pioctl* call. The **input parameter** refers to the storage passed in *blob.in* and the **output parameter** refers to the storage passed in *blob.out*.

For example, the following call to the *pioctl* system call sets an access control list on *"/afs/andrew/usr/kazar"* to allow anyone to read the directory.

Note that for those *pioctls* that do not affect a file, one should provide a pathname parameter of NULL (0).

```
struct ViceIoctl blob;
char *acl = "1\n0\nSystem:AnyUser      127\n";
blob.in_size = strlen(acl)+1; /* don't forget the null */
blob.in = acl;
blob.out_size = 0; /* no out parameters */
code = pioctl("/afs/andrew/usr/kazar", VIOCSETAL, &blob, 1);
if (code < 0) perror("setacl");
```

3.2.1. VIOCSETAL

Set access control list. Input parameter is an access control list; there are no output parameters. The access control list on the directory specified by the pathname parameter to the *pioctl* is set to the input parameter. EINVAL will be returned if the access list string is not properly formatted. You must have administer rights to a directory, own the directory, or be a system administrator to change the access list associated with a directory. Access control lists are represented as ASCII character strings, each containing several lines. The first line is the count of the number of positive access control list entries, represented as a character string. The second line gives the number of negative access control list entries. The remaining lines, each terminated with a "\n", give the actual access control list entries. Each entry line consists of two fields, separated by a space. The first is the group or user name whose access to the directory is being given. The second is the access rights, represented as a decimal number. Each access right is represented by one bit in the access right integer; the logical sum of these gives the value for a set of access rights. The symbolic definitions for the access rights are given in the file */usr/andy/include/afs/prs_fs.h*.

3.2.2. VIOCGETAL

Get access control list. There are no input parameters; the one output parameter is the access control list of the directory specified by the first parameter to the *pioctl* system call. The access control list format is described above under *VIOCSETAL*.

3.2.3. VIOCSETTOK

Set authentication tokens. This call has one input parameter, containing the authentication tokens, encoded as described below. The authentication tokens provided are associated with the Unix UID of the process performing the call. These tokens are obtained (usually by *log* or *login*) by contacting the (Kerberos) authentication server, using the user's password as an encryption key. These tokens can be used to authenticate with any file server. Tokens come in pairs; a **clear token** and a **secret token**. The secret token is encrypted (by the Authentication Server) with the file server's key, and can be safely transmitted over the network. The clear token contains unencrypted information from the secret token and should not be transmitted over the network.

The input parameter encodes these structures as follows.

The first 4 bytes give the size of the secret token.

The next `sizeof(struct SecretToken)` bytes contain the secret token itself.

The next 4 bytes give the size of the clear token.

The next `sizeof(struct ClearToken)` bytes are the clear token itself.

Each field should be byte-aligned; fields are copied around using *bcopy*. If the tokens are not valid, the code EPERM will be returned. EINVAL will be returned if the clear token size is incorrect. Like *VIOCGETTOK*, this call has also been extended to support multiple cells. In particular, one may provide extra information after the tokens, which is interpreted to be a long integer specifying whether this is the primary cell ID for the user, and a character string giving the cell name for the tokens. This call will fail if the cell has not yet been configured to this cache manager, with error code ESRCH. If no primary ID and cell string values are provided to this call, AFS sets the tickets for the workstation's home cell, and assumes that the primary cell flag is 0.

3.2.4. VIOCGETVOLSTAT

Get volume status. This call has no input parameter, and has one output parameter, which

contains four fields, concatenated using *bcopy*.

The output parameter's first field is a *struct VolumeStatus* structure (defined in `/usr/andy/include/afs/afsint.h`). It is immediately followed by the volume name, of maximum length 32, terminated with a single null character. The volume name is followed by another character string, of maximum length 256, terminated with a single null character, containing the offline message associated with the volume. Operators may put messages having to do with why a volume is offline in this field. The last field is the volume's message of the day, a character string of maximum length 256, terminated by a null character. Operator's may put any message of interest to users of the volume in this field.

As with all *pioctl* parameters, the various fields are only byte-aligned.

3.2.5. VIOCSETVOLSTAT

Set volume status. This call sets the volume's status information. You must be authenticated as a system administrator to set any of the status fields of a volume, such as the quotas associated with a volume. This call has one input parameter and no output parameters.

The input parameter contains a *struct VolumeStatus* structure, followed by the volume name, offline message, and message of the day, all encoded as described under the *VIOCGETVOLSTAT* call above.

3.2.6. VIOCFLUSH

Invalidate cache. This call has no input or output parameters. This call attempts to remove any cached information associated with the file named in the *pathname* parameter to the *pioctl* call. The next reference to the file will fetch this information again from the File Server.

3.2.7. VIOCSTAT

List contents of a mount point. This call is used to get information about an Andrew File System mount point. These objects, not to be confused with Unix file system mount points, are Andrew File System objects that indicate to the Andrew File System where a particular volume is supposed to appear in the file system tree. The *pathname*

parameter specifies the mount point. This call has no input parameter. The output parameter gives the contents of the mount point:

The mount point consists of a character string giving the type of mount point, followed by the name of the volume to be mounted at this point in the tree. The name can optionally be given as the ascii representation of the volume's number. The type character can be either '%' or '#':

- A type of '#' tells the file system to use the latest read-only volume corresponding to the named volume, if the mount point is part of a read-only file system. Otherwise the mount point functions as a mount point to the named volume.
- A type of '%' is normal: the mount point simply names the volume to be mounted at this point in the file system.

The mount point string is not null-terminated.

3.2.8. VIOCGETTOK

Get authentication tokens. This call gets the authentication tokens associated with the UID making the *pioctl* call. This call has two forms, a long form and a short form. In the short form, this call has no input parameters. Its one output parameter, the authentication tokens, are encoded as in the *VIOCSETTOK* call. In the long form, this call supports multiple cells. The long form call takes one input parameter, a long integer, between 0 and infinity, telling the Cache Manager which set of tokens to return. There is no connection between this number and any other cell number; it is just an iterator mechanism for performing multiple calls to the Cache Manager to get the required information. When using the new calling convention, the call returns a long integer after the token information returned by the short form call, indicating whether this is the primary cell id, and a character string indicating the cell name. When using the long form, this call returns the error code *EDOM* when there are no more tokens to retrieve.

3.2.9. VIOCUNLOG

Discard tokens. This call tells the Cache Manager to discard the authentication tokens associated with the Unix UID of the process making the call. This call is generally done before logging out, to prevent unauthorized people from making use of one's authentication tokens. This call has no input and no output parameters.

3.2.10. VIOCCKSERV

Check servers. This call directs the file system client process to check immediately which servers are running. Normally the client process checks dead servers every four minutes to see whether they are back up; otherwise it checks working servers every 10 minutes to make sure they are still running. This call has one optional input parameter and no output parameters. The input parameter, if specified, is treated as an integer. If this value is 1, then the cache manager does not check the server status at this time, but instead only reports the status of the servers as currently known.

3.2.11. VIOCCKBACK

Check backups. This call directs the file system client process to re-evaluate volume names to see if more recent volumes with the same name have been created. Normally the cache manager executes this function every 2 hours. This call has no input and no output parameters.

3.2.12. VIOCCKCONN

Check authentication. This call checks that all connections for the Unix UID making the *pioctl* call are properly authenticated. It has no input parameter and one output parameter. The output parameter is an integer whose value is 0 if all connections are authenticated and 1 otherwise.

3.2.13. VIOCGETTIME

Reserved. This call is only used for performance timings, and may change without notice.

3.2.14. VIOCWHEREIS

Locate file. This call returns an array of 8 32-bit internet addresses (in network byte order). These addresses give the location, or locations for replicated files, of the volume containing the file named by the pathname parameter. Unused entries are set to 0.

3.2.15. VIOCEPREFETCH

Pre-fetch file. The file named by the pathname parameter to the call is fetched into the cache. The call returns before the fetch completes. This call can be used when a program knows it will shortly require use of the data in a file.

3.2.16. VIOCNOP

Reserved. Another call reserved for timings.

3.2.17. VIOCENGROUP

This call is obsolete.

3.2.18. VIOCDISGROUP

This call is obsolete

3.2.19. VIOCLISTGROUPS

This call is obsolete

3.2.20. VIOACCESS

The Andrew File System's equivalent of the Unix *access* call. While the Andrew Cache Manager tries to interpret an *access* system call in a reasonable fashion, certain protection features of the Andrew File System do not map cleanly onto the Unix protection mechanism. For instance, in the Unix file system, you can use the *access* system call to determine whether you can insert or delete a file from a directory by checking "write" permission to that directory. In the Andrew File System, two independent rights control modifications to a directory: insert rights allow the addition of files to a directory, and delete rights allow the removal of files from a directory. The *access* system call reports that a user can write a directory if that user can do either of these operations. If you need more detailed information, use this call.

This call has one input parameter and no output parameters. The one input parameter is an integer (passed as 4 bytes), representing the access modes to be checked against the file whose name was passed as the first parameter to the *pioctl*. More than one access bit can be turned on in the input parameter: the call will check that the requesting user can perform all of the requested operations. The individual protection bits are defined in the file `"/usr/andy/include/afs/prs_fs.h"`.

If the access requested is allowed by the current mode bits on the file, and the

appropriate directory's access control list, then the *pioctl* call will succeed and return a value of 0. Otherwise, it will return -1 and the global variable `errno` will be set to `EACCES`. If the file does not

exist, or for some other reason cannot be accessed, the pioctl call will fail and errno will be set to indicate the reason for the error.

3.2.21. VIOCUNPAG

This call performs the same function as *VIOCUNLOG*. They both exist only for historical reasons.

3.2.22. VIOCGETWD

This call is obsolete

3.2.23. VIOCWAITFOREVER

This call is obsolete

3.2.24. VIOCSETCACHESIZE

Tell the cache manager how much space to use for its cache. One input parameter, a long integer representing the amount of space in 1K byte units. No output parameters. A value of 0 tells the cache manager to reset its cache limit to the limit set initially at boot time.

3.2.25. VIOCFLUSHCB

Tell the cache manager to flush a callback, but not the data, associated with the file or directory specified by the pathname parameter. No input or output parameters.

3.2.26. VIOCNEWCELL

Tell the cache manager to configure a new cell. The input parameter contains MAXHOSTS(8) network order hosts, possibly null-terminated (if there are less than 8 hosts) that represent the volume location servers for this cell. This 32-byte array is followed by the null-terminated cell name.

3.2.27. VIOCGETCELL

Get the cell name and hosts. Takes one input parameter, a zero-based iterator. Returns the same structure as *VIOCNEWCELL* provides. When the index gets too large (i.e. exceeds the number of cells in the system), this call returns a standard Unix error (EDOM).

3.2.28. VIOC_AFS_DELETE_MT_PT

Delete a mount point. The pathname identifies the directory containing the mount point to be removed. The input parameter is a null-terminated character string giving the entry name within the specified directory naming the mount point to be removed. This call returns ENOTDIR if the directory pathname does not in fact specify a directory. This call has no output parameters.

3.2.29. VIOC_AFS_STAT_MT_PT

Return information about a mount point. As in the *VIOC_AFS_DELETE_MT_PT* call, the pathname specifies the directory containing the mount point, and the input parameter is a null-terminated character string representing the entry name of the mount point. This call returns one output parameter, a null-terminated character string containing the mount point's representation. This consists of a single type character (consisting of either a "#" or a "%" character), followed by an optional cell name followed by a colon (":") character (present only if the cell name is present). Finally, the mount point ends with the volume name.

3.2.30. VIOC_FILE_CELL_NAME

Given a file name, determine the name of the cell in which that file resides. This call has no input parameters, and returns a character string identifying the file's corresponding cell. The file is specified by the pathname parameter. If the file doesn't exist, or for

some other reason cannot be accessed, the call will fail and *errno* will indicate the reason for the error.

3.2.31. VIOC_GET_WS_CELL

Get the name of the cell to which the workstation belongs. This call has no input parameters, and returns a character string identifying the ``home cell'' for the workstation.

3.2.32. VIOC_AFS_MARINER_HOST

This call sets the host to which the cache manager should direct *mariner* information. This call has one input parameter, a long integer, and one output parameter, another long integer. The input parameter is either the internet host address (in network byte order) of the host to which to send mariner fetch and store messages, or 0xffffffff, meaning that mariner messages should not be generated, or 0, which means do not change the mariner host address. The output parameter is set to the previous value of the mariner host field. The mariner messages are UDP messages sent to UDP/IP port 2106 on the mariner host.

3.2.33. VIOC_GET_PRIMARY_CELL

Return the name of the cell in which the caller has established his ``primary'' identity (i.e. where the caller has logged in). Note: the *log* program can be used to establish non-primary tokens of identity in any number of cells; these tokens are not reported here. If the user has done an *unlog*, then he or she will not have a primary identity, and this call will return the null string.

3.2.34. VIOC_VENUSLOG

Control venus logging (kernel venus only). One input parameter, a long integer, interpreted as follows. The low order byte controls whether a log file is written; a one (1) means that a log file is started, a zero (0) ends a log file, and 99 means return the equivalent input parameter to set the current logging parameters. The next higher-order byte (the byte shifted left by 8 bits from the low-order byte) sets the RFTP debug level; a value of 99 means do not change this value. The next highest-order byte (the byte shifted left by 16 bytes) controls the RPC debug level, where again, a value of 99 means not to change the value. Finally, the highest-order byte (the one shifted left by 24 bits) is the AFS debug level; a 99 means not to change the value. If the value of the low-order byte is 99, then this call returns an output parameter, a long integer, giving updated debug levels. The returned long integer has the same format as the input parameter to this call (except, of course, that a value of 99 for a subfield is never returned).

3.2.35. VIOC_GETCELLSTATUS

This call has one input parameter, the character string name of a configured cell. It returns two long integers. The first is the cell status, whose meaning is defined in *afs.h*. The second is reserved for future use, and at present is 0. The cell must already be configured, or the call will fail, setting *errno* to ENOENT.

3.2.36. VIOC_SETCELLSTATUS

This call has three input parameters. The first parameter is the new cell status field (a long integer), whose bit fields are defined in *afs.h*. The second parameter is a long integer reserved for future use, and must be 0. The third input parameter is the name of the cell whose status will be set. The cell is represented as a null-terminated

character string, and should already be configured, or the call will fail, setting *errno* to ENOENT.

3.3. Complete list of AFS Assigned Numbers Registry - *ioctl()* and *pioctl()* function codes

3.3.1. COMMON IOCTL() CODES

Group	Code	Name	Purpose / Usage
C	0	- reserved -	
C	239	- reserved -	Reserved for code space expansion
C	255	- reserved -	Reserved for code space expansion

3.3.2. TRANSARC/IBM AND LEGACY IOCTL() CODES

Group	Code	Name	Purpose / Usage
V	0	- reserved -	
V	1	VIOCCLOSEWAIT	Force close() to wait for store
V	2	VIOCABORT	Abort close on this fd
V	3	VIOCIGETCELL	Get cell name
V	239	- reserved -	Reserved for code space expansion
V	240	VIOC_SITE1	Reserved for site-specific extensions
V	241	VIOC_SITE2	Reserved for site-specific extensions
V	242	VIOC_SITE3	Reserved for site-specific extensions
V	243	VIOC_SITE4	Reserved for site-specific extensions
V	244	VIOC_SITE5	Reserved for site-specific extensions
V	245	VIOC_SITE6	Reserved for site-specific extensions
V	246	VIOC_SITE7	Reserved for site-specific extensions
V	247	VIOC_SITE8	Reserved for site-specific extensions
V	248	VIOC_PRIVATE1	Reserved for implementation use
V	249	VIOC_PRIVATE2	Reserved for implementation use
V	250	VIOC_PRIVATE3	Reserved for implementation use
V	251	VIOC_PRIVATE4	Reserved for implementation use
V	252	VIOC_PRIVATE5	Reserved for implementation use

V	253	VIOC_PRIVATE6	Reserved for implementation use
V	254	VIOC_PRIVATE7	Reserved for implementation use
V	255	VIOC_PRIVATE8	Reserved for implementation use

3.3.3. COMMON PIOCTL() CODES

Group	Code	Name	Purpose / Usage
C	0	- reserved -	
C	1	VIOC_NEWALIAS	Add new cell alias
C	2	VIOC_GETALIAS	Get cell alias info
C	3	VIOC_CBADDR	Push callback address
C	4	VIOC_CHOWN	Change file owner
C	5	VIOC_DISCON	Set/get diconnected mode
C	6	VIOC_CREATE_MT_PT	Create mount point
C	7	VIOCGETTOK2	Get tokens (extended)
C	8	VIOCSETTOK2	Set tokens (extended)
C	9	VIOCNEWUUID	Force a new uuid
C	10	VIOCGETPROP	Get property
C	11	VIOCSETPROP	Set property
C	239	- reserved -	Reserved for code space expansion
C	255	- reserved -	Reserved for code space expansion

3.3.4. TRANSARC/IBM AND LEGACY PIOCTL() CODES

Group	Code	Name	Purpose / Usage
V	0	- reserved -	
V	1	VIOCSETAL	Set access control list
V	2	VIOCGETAL	Get access control list
V	3	VIOCSETTOK	Set authentication tokens
V	4	VIOCGETVOLSTAT	Get volume status
V	5	VIOCSETVOLSTAT	Set volume status
V	6	VIOCFLUSH	Invalidate cache entry
V	7	VIOCSTAT	Get file status
V	8	VIOCGETTOK	Get authentication tokens

V	9	VIOCUNLOG	Invalidate tokens
V	10	VIOCKSERV	Check that servers are up
V	11	VIOCKBACK	Check backup volume mappings
V	12	VIOCKCONN	Check connections for a user
V	13	VIOCGETTIME	Make gettime call (historic)
V	14	VIOCWHEREIS	Find out where a volume is located
V	15	VIOCPREFETCH	Prefetch a file
V	16	VIOCNOP	No-op (historic)
V	17	VIOCENGROUP	Enable group access (historic)
V	18	VIOCDISGROUP	Disable group access (historic)
V	19	VIOCLISTGROUPS	List enabled and disabled groups (historic)
V	20	VIOACCESS	Access using PRS_FS bits
V	21	VIOCUNPAG	Invalidate pag
V	22	VIOCGETFID	Get file ID quickly
V	23	VIOCWAITFOREVER	Wait for dead servers forever (historic)
V	24	VIOCSETCACHESIZE	Set venus cache size in 1k units
V	25	VIOCFLUSHCB	Flush callback only
V	26	VIOCNEWCELL	Configure new cell
V	27	VIOCGETCELL	Get cell info
V	28	VIOC_AFS_DELETE_MT_PT	Delete mount point
V	29	VIOC_AFS_STAT_MT_PT	Stat mount point
V	30	VIOC_FILE_CELL_NAME	Get cell in which file lives
V	31	VIOC_GET_WS_CELL	Get cell in which workstation lives
V	32	VIOC_AFS_MARINER_HOST	Get/set mariner (cache manager monitor) host
V	33	VIOC_GET_PRIMARY_CELL	Get primary cell for caller
V	34	VIOC_VENUSLOG	Enable/Disable venus logging
V	35	VIOC_GETCELLSTATUS	Get cell status info
V	36	VIOC_SETCELLSTATUS	Set corresponding info
V	37	VIOC_FLUSHVOLUME	Flush whole volume's data
V	38	VIOC_AFS_SYSNAME	Change @sys value
V	39	VIOC_EXPORTAFS	Export afs to nfs clients
V	40	VIOCGETCACHEPARMS	Get cache stats

V	41	VIOCGETVCXSTATUS	
V	42	VIOC_SETSPREFS33	Set server ranks (deprecated)
V	43	VIOC_GETSPREFS	Get server ranks
V	44	VIOC_GAG	Silence CM
V	45	VIOC_TWIDDLE	Adjust RX knobs
V	46	VIOC_SETSPREFS	Set server ranks
V	47	VIOC_STORBEHIND	Adjust store asynchrony
V	48	VIOC_GCPAGS	Disable automatic pag gc-ing
V	49	VIOC_GETINITPARAMS	Get initial cm params
V	50	VIOC_GETCPREFS	Get client interface
V	51	VIOC_SETCPREFS	Set client interface
V	52	VIOC_AFS_FLUSHMOUNT	Flush mount symlink data
V	53	VIOC_RXSTAT_PROC	Control process RX stats
V	54	VIOC_RXSTAT_PEER	Control peer RX stats
V	55	VIOC_GETRXKCRYPT	Get rxkad encryption flag
V	56	VIOC_GETRXKCRYPT	Set rxkad encryption flag
V	57	VIOC_FPRIOSTATUS	arla: set file prio
V	58	VIOC_FHGET	arla: fallback getfh
V	59	VIOC_FHOPEN	arla: fallback fhopen
V	60	VIOC_XFSDEBUG	arla: controls xfsdebug
V	61	VIOC_ARLADEBUG	arla: controls arla debug
V	62	VIOC_AVIATOR	arla: debug interface
V	63	VIOC_XFSDEBUG_PRINT	arla: print xfs status
V	64	VIOC_CALCULATE_CACHE	arla: force cache check
V	65	VIOC_BREAKCELLBACK	arla: break callback
V	99	PSetClientContext	NFS-pag pioctl (for knfs/rmtsysd)
V	239	- reserved -	Reserved for code space expansion
V	240	VIOC_SITE1	Reserved for site-specific extensions
V	241	VIOC_SITE2	Reserved for site-specific extensions
V	242	VIOC_SITE3	Reserved for site-specific extensions

V	243	VIIOC_SITE4	Reserved for site-specific extensions
V	244	VIIOC_SITE5	Reserved for site-specific extensions
V	245	VIIOC_SITE6	Reserved for site-specific extensions
V	246	VIIOC_SITE7	Reserved for site-specific extensions
V	247	VIIOC_SITE8	Reserved for site-specific extensions
V	248	VIIOC_PRIVATE1	Reserved for implementation use
V	249	VIIOC_PRIVATE2	Reserved for implementation use
V	250	VIIOC_PRIVATE3	Reserved for implementation use
V	251	VIIOC_PRIVATE4	Reserved for implementation use
V	252	VIIOC_PRIVATE5	Reserved for implementation use
V	253	VIIOC_PRIVATE6	Reserved for implementation use
V	254	VIIOC_PRIVATE7	Reserved for implementation use
V	255	VIIOC_PRIVATE8	Reserved for implementation use

Appendice B -Variabili di ambiente disponibili nell'ambiente di esecuzione dell'esub

LSB_SUB_PARM_FILE	This variable points to a temporary file containing the job parameters that esub reads when the job is submitted. The submission parameters are a set of name-value pairs on separate lines in the format "option_name=value". The following option names are supported:	LSB_SUB_ADDITIONAL	String format parameter containing the value of the -a option to bsub. The value of -a is passed to esub, but it does not directly affect the other bsub parameters or behavior. The value of -a must correspond to an actual esub file. For example, to use bsub -a fluent, the file esub.fluent must exist in LSF_SERVERDIR. LSB_SUB_ADDITIONAL cannot be changed in or added to SB_SUB_MODIFY_FILE.
		LSB_SUB_BEGIN_TIME	Begin time, in seconds since 00:00:00 GMT, Jan. 1, 1970
		LSB_SUB_CHKPNT_DIR	Checkpoint directory
		LSB_SUB_COMMAND_LINE	bsub job command argument
		LSB_SUB_COMMANDNAME	must be set in lsf.conf to enable esub to use this variable
		LSB_SUB_CHKPNT_PERIOD	Checkpoint period
		LSB_SUB_DEPEND_COND	Dependency condition
		LSB_SUB_ERR_FILE	Standard error file name
		LSB_SUB_EXCEPTION	Exception condition
		LSB_SUB_EXCLUSIVE	"Y" specifies exclusive execution
		LSB_SUB_EXTSCHED_PARAM	Validate or modify bsub -extsched option
		LSB_SUB_HOST_SPEC	Host specifier
		LSB_SUB_HOSTS	List of execution host names
		LSB_SUB_IN_FILE	Standard input file name
		LSB_SUB_INTERACTIVE	"Y" specifies an interactive job
		LSB_SUB_LOGIN_SHELL	Login shell
		LSB_SUB_JOB_NAME	Job name

		LSB_SUB_JOB_WARNING_ACTION	Job warning action specified by bsub -wa
		LSB_SUB_JOB_WARNING_TIME_PERIOD	Job warning time period specified by bsub -wt
		LSB_SUB_MAIL_USER	Email address used by LSF for sending job email
		LSB_SUB_MAX_NUM_PROCESSORS	Maximum number of processors requested
		LSB_SUB_MODIFY	"Y" specifies a modification request
		LSB_SUB_MODIFY_ONCE	"Y" specifies a modification-once request
		LSB_SUB_NOTIFY_BEGIN	"Y" specifies email notification when job begins
		LSB_SUB_NOTIFY_END	"Y" specifies email notification when job ends
		LSB_SUB_NUM_PROCESSORS LSB_SUB_OTHER_FILES	Minimum number of processors requested Always "SUB_RESET" if defined to indicate a bmod is being performed to reset the number of files to be transferred
		LSB_SUB_OTHER_FILES_number	number is an index number indicating the particular file transfer value is the specified file transfer expression. For example, for bsub -f "a > b" -f "c < d", the following would be defined: LSB_SUB_OTHER_FILES_0="a > b" LSB_SUB_OTHER_FILES_1="c < d"
		LSB_SUB_OUT_FILE	Standard output file name
		LSB_SUB_PRE_EXEC	Pre-execution command
		LSB_SUB_PROJECT_NAME	Project name
		LSB_SUB_PTY	"Y" specifies an interactive job with PTY support
		LSB_SUB_PTY_SHELL	"Y" specifies an interactive job with PTY shell support
		LSB_SUB_QUEUE	Submission queue name
		LSB_SUB_RERUNNABLE	"Y" specifies a rerunnable job
		LSB_SUB_RES_REQ	Resource requirement string
		LSB_SUB_RESTART	"Y" specifies a restart job
		LSB_SUB_RESTART_FORCE	"Y" specifies forced restart job
		LSB_SUB_RLIMIT_CORE	Core file size limit
		LSB_SUB_RLIMIT_CPU	CPU limit
		LSB_SUB_RLIMIT_DATA	Data size limit
		LSB_SUB_RLIMIT_FSIZE	File size limit
		LSB_SUB_RLIMIT_RSS	Resident size limit
		LSB_SUB_RLIMIT_RUN	Wall-clock run limit
		LSB_SUB_RLIMIT_STACK	Stack size limit
		LSB_SUB_RLIMIT_THREAD	Thread limit

		LSB_SUB_TERM_TIME	Termination time, in seconds, since 00:00:00 GMT, Jan. 1, 1970
		LSB_SUB_TIME_EVENT	Time event expression
		LSB_SUB_USER_GROUP	User group name
		LSB_SUB_WINDOW_SIG	Window signal number
		LSB_SUB2_JOB_GROUP	Options specified by bsub -g
		LSB_SUB2_LICENSE_PROJECT	LSF License Scheduler project name specified by bsub -Lp
		LSB_SUB2_SLA	SLA scheduling options
		LSB_SUB2_USE_RSV	Advance reservation ID specified by bsub -U
LSB_SUB_ABORT_VA LUE	This variable indicates the value esub should exit with if LSF is to reject the job submission.		
LSB_SUB_MODIFY_E NVFILE	The file in which esub should write any changes to the job environment variables. esub writes the variables to be modified to this file in the same format used in LSB_SUB_PARM_FILE. The order of the variables does not matter. After esub runs, LSF checks LSB_SUB_MODIFY_ENVFILE for changes and if found, LSF will apply them to the job environment variables.		
LSB_SUB_MODIFY_FI LE	The file in which esub should write any submission parameter changes. esub writes the job options to be modified to this file in the same format used in LSB_SUB_PARM_FILE. The order of the options does not matter. After esub runs, LSF checks LSB_SUB_MODIFY_FILE for changes and if found LSF will apply them to the job. LSB_SUB_ADDITIONAL cannot be changed in or added to LSB_SUB_MODIFY_FILE.		
LSF_INVOKE_CMD	Indicates the name of the last LSF command that invoked an external executable (for example, esub). External executables get called by several LSF commands (bsub, bmod, lsrn). This variable contains the name of the last LSF command to call the executable.		

Appendice C -Compilazione ed installazione di OpenAFS in ambiente Windows

***** Windows 2000/XP/2003 Build OpenAFS Process *****

- Testato su Windows XP Service Pack2
- OpenAfs 1.5.24
- Microsoft Visual Studio 2003 / Microsoft Visual Studio 2005
- Microsoft Platform SDK per Windows XP SP2 or Server 2003 SP1 or Vista
- Microsoft Windows DDK per Server 2003 SP1
- The Microsoft HTML Help Workshop
- Wix 2.0.2217.0

-- NB: seguire sempre come guida di riferimento il file README-NT ufficiale --

Passo 1:

Dopo aver ottenuto il pacchetto con i sorgenti di OpenAfs (www.openafs.org), scompattarlo in una directory del disco

Es:

```
C:\Programmi\openafs-1.5.24-src\openafs-1.5.24
```

Copiare i file 'NTMakefile' e 'ntbuild.bat' dalla cartella 'src' della directory base di OpenAfs (che da questo momento chiameremo %AFSROOT%):

Quindi da una shell di dos eseguiamo:

```
cd Programmi\OpenAFS\OpenAFS-1.5.24
copy src\NTMakefile .
copy src\ntbuild.bat .
```

Passo 2:

Configurazione del file ntbuild.bat.

SYS_NAME: Scegliere tra "i386_w2k", "amd64_w2k", oppure "i64_w2k" a seconda del processore

_WIN32_IE: Settare a "0x500" per Windows 2000 e superiori
Settare a "0x400" per Win9x/NT4

AFSROOT: Specificare la directory base di OpenAFS (8.3 short name)
(es: "c:\progra~1\OPENAF~1.24-\OPENAF~1.24")

AFSVER_CL: Settare a "1200" se si usa MS Visual C++ 6.0
Settare a "1300" se si usa MS Visual Studio .NET
Settare a "1310" se si usa MS Visual Studio .NET 2003
Settare a "1400" se si usa MS Visual Studio .NET 2005

MSVCDIR: Locazione della cartella di Microsoft Visual C++ (8.3 short name)
(es: "c:\progra~1\MID05A~1\VC")

MSSDKDIR: Locazione di Microsoft Platform SDK (8,3 short name)
(es: "c:\progra~1\MIC977~1")

NTDDKDIR: Locazione di %DDK%\inc\wnet

(es: "C:\WinDDK\3790~1.183\inc\wnet")

NTDDKDIR2: Locazione di %DDK%\inc\crt

(es: "C:\WinDDK\3790~1.183\inc\crt")

Passo 3 (Opzionale):

C'è la possibilità aggiungere un file che contiene le impostazioni delle varie celle, ed è possibile scaricarlo all'indirizzo:

<http://grand.central.org/dl/cellservdb/CellServDB>

Copiarlo nella directory %AFSROOT%\src\WINNT\install\NSIS e rinominarlo in a"fsdcell.ini".

Modificare il file %AFSROOT%\src\config\NTMakefile.i386_w2k

AFSPRODUCT_VER_MAJOR	Version Major Number
AFSPRODUCT_VER_MINOR	Version Minor Number
AFSPRODUCT_VER_PATCH	Version Patch Number
AFSPRODUCT_VER_BUILD	Version Build Number
CELLSERVDB_INSTALL	The default file name for the CellServDB included in the install Package.
CELLNAME_DEFAULT	The default home cell name.
CELLSERVDB_WEB	The default web address to obtain CellServDB

Es:

```
AFSPRODUCT_VER_MAJOR=1
AFSPRODUCT_VER_MINOR=5
AFSPRODUCT_VER_PATCH=2400
AFSPRODUCT_VER_BUILD=0
CELLSERVDB_INSTALL=CellServDB.GrandCentral
CELLNAME_DEFAULT=enea.it
CELLSERVDB_WEB=http://grand.central.org/dl/cellservdb/CellServDB
```

Passo 4 (Compilazione con Visual Studio 2003):

Prima della compilazione posizionarsi nella cartella %AFSROOT%\src\config. Cercare il file "Makefile.version-CML", effettuarne una copia e rinominarla in "Makefile.version".

- Dalla shell di DOS posizionarsi in %AFSROOT%;
- Eseguire il comando "vsvars32.bat", solitamente nella directory di Microsoft Visual Studio; (Nel nostro caso "C:\Programmi\Microsoft Visual Studio 8\Common7\Tools");
- Eseguire il comando "SetEnv.cmd /2000 /DEBUG", solitamente nella directory di Microsoft Platform SDK;
- Eseguire il comando "ntbuild.bat checked";
- Eseguire il comando "nmake /f NTMakefile clean";
- Eseguire il comando "nmake /f NTMakefile install".

Se tutto è andato a buon fine nmake terminerà con l'output "Build Finished Successfully" altrimenti con un error code di tipo 0x2.

Passo 5 (Installazione di Wix con Visual Studio 2005):

Scaricare il pacchetto Wix da <http://prdownloads.sourceforge.net/wix/sources-2.0.2217.0.zip>

Applicare tutte la patch spiegate nel file README-NT e lanciare il comando "make ship" dalla directory dei sorgenti di Wix (%WIXROOT\src).

NOTA: la compilazione del pacchetto wix non avviene in modo corretto utilizzando Visual Studio 2003 poiché non riesce ad interpretare alcuni tipi di file definiti in "winnt.h", quindi stiamo utilizzando Visual Studio 2005.

Prima di cominciare la compilazione di wix dobbiamo applicare delle modifiche ad alcuni file.

Aprire il file "comutil.h" di "%WINDDK%\inc\wnet" e alla riga 17 modificare il path di inclusione del file intsafe.h in "../crt/intsafe.h".

Aprire il file NTMakefile di "%AFSROOT%\src\WINNT\install\loopback" e alla riga 56 aggiungere /NODEFAULTLIB: "libc.lib" subito dopo "\$ (LINK)".

Aprire il file "property.wxi" di "%AFSROOT%\src\WINNT\install\wix" ed eliminare tutte le diciture Type=raw

Aprire il file "ui.wxi" di "%AFSROOT%\src\WINNT\install\loopback\lang\en_US" e sostituire tutti gli elementi "RadioButtonGroup" con "RadioGroup". Poi eliminare da ognuno di essi l'attributo "Value" e inserire uno 0 (zero) e un 1 come elementi a seconda di come l'attributo "Value" era settato.

Esempio:

PRIMA:

```
<RadioButtonGroup Property="SECURITYLEVEL">
  <RadioButton X="5" Y="0" Width="55" Height="15" Text="Disable" Value="0">
  </RadioButton>
  <RadioButton X="65" Y="0" Width="55" Height="15" Text="Enable" Value="1">
  </RadioButton>
</RadioButtonGroup>
```

DOPO:

```
<RadioGroup Property="FREELANCEMODE">
  <RadioButton X="5" Y="0" Width="55" Height="15" Text="Disable">0</RadioButton>
  <RadioButton X="65" Y="0" Width="55" Height="15" Text="Enable">0</RadioButton>
</RadioGroup>
```

Aprire il file "files.wxi" di "%AFSROOT%\src\WINNT\install\wix" e posizionarsi alla riga 776. Modificare l'attributo "src" eliminando '\' dall'inizio del path e aggiungendone uno alla fine.

Posizionarsi nella cartella "%AFSROOT%\src\WINNT\netidmgr_plugin" e lanciare il comando "hhc afsplhlp.hhp" che creerà il file afsplhlp.chm. Spostare questo file nella directory "%AFSROOT%\dest\i386_w2k\checked\root.client\usr\vice\etc".