Laboratorio di Programmazione di Rete

Progetto conclusivo per l'Anno Accademico 2007-2008

File system sicuro

M. Danelutto - Dicembre 2007

Versione 1.1

Scopo

Lo scopo del progetto è quello di mettere a disposizione degli utenti un *sistema per la memorizzazione distribuita e ridondante di file importanti*. In particolare "distribuita" significa che i file dovranno essere memorizzati si macchine raggiungibili via rete e "ridondante" significa che dovranno essere mantenute, dal sistema, una serie di copie su macchina diverse in modo che, nel caso di un guasto ad una macchina o ad un disco si possa comunque recuperare una versione del file.

L'accesso ai file memorizzati secondo questa procedura deve avvenire nel modo più trasparente possibile. Si dovrà quindi definire un insieme di classi (tipicamente una per i file in lettura ed una per i file in scrittura) che permettano l'accesso al sistema di memorizzazione distribuita e ridondante semplicemente dichiarando un oggetto "file remoto" ed utilizzandone i medodi per la lettura, scrittura e chiusura, esattamente come avremmo fatto se il file fosse stato un normale file del disco locale utilizzando oggetti di tipo FileInputStream o FileOutputStream. L'accesso ai file del file system sicuro dovrà garantire che un file possa essere aperto in lettura da un numero arbitrario di processi lettori, anche da macchine diverse, ma che possa essere aperto in scrittura da un solo processo scrittore e che tale apertura precluda la possibilità di aprire lo stesso file in lettura da parte di altri processi finchè le operazioni di scrittura non siano effettivamente terminate.

Per semplificare l'implementazione, si potrà assumere che la modalità "aperto in scrittura" per un file corrisponda alla creazione del file stesso e che, nel caso il file sia già presente nel file system sicuro, la vecchia copia venga cancellata o ridenominata con un suffisso ".orig". In questo secondo caso, ovviamente, alla seconda riapertura in scrittura del file la copia precedente verrà soprascritta e quindi il meccanismo implementa di fatto una sola copia di backup. Per la stessa motivazione, non è richiesto che il file sistem sicuro implementi una gerachia di directory. Tutti i file aperti utilizzando il sistema saranno allo stesso livello, Quindi lo spazio dei nomi è *flat*.

Il sistema dovrà essere configurabile, in modo che il numero delle copie da mantenere per ciascuno dei file sia un parametro definito a tempo di compilazione. Tutti i serventi necessari per l'implementazione del file system sicuro dovranno essere lanciati indipendentemente dal lancio del programma (o dei programmi) che utilizza(no) il file system sicuro. La localizzazione dei serventi con cui le classi che implementaton i "file remoti" dovrà avvenire senza alcuna conoscenza della loro effettiva ubicazione, ovvero andranno previsti opportuni meccanismi di discovery.

Infine, per migliorare le caratteristiche di sicurezza del file system sicuro, si dovranno prevedere almeno due tipi di server: i server “file system”, che provvedono alla gestione dei file memorizzati nel file system sicuro e alla implementazione delle richieste dei processi utente, e i server “disco” che provvedono semplicemente alla memorizzazione sul file system locale, un una sottodirectory della “/tmp” configurata a tempo di compilazione, dei file del file system sicuro, secondo le richieste generate da serer di tipo “file system”.

Lo schema generare di funzionamento dell’intero sistema sarà dunque tipo quello della Figura 1.

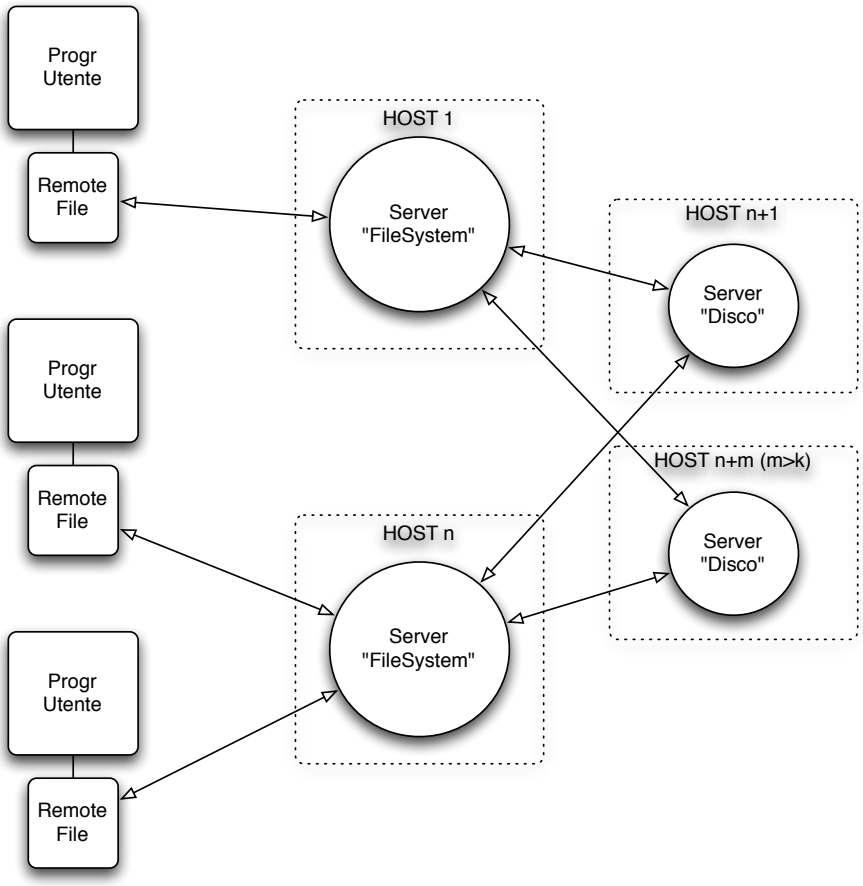

Figura 1: schema generale del file system sicuro

Si noti come siano le sole classi che implementano i remote file ad interagire con il sistema, che possono esistere più server di tipo file system, per ragioni di tolleranza ai guasti (un unico server rappresenterebbe un *single point of failure* molto pericoloso), che i server file system gestiscono uno stesso insieme di server disco e che di conseguenza lo spazio dei nomi dei file memorizzati nel file system sicuro è unico e condiviso fra i vari serventi file system.

Classi di interfaccia

Per accedere al file system sicuro si devono utilizzare due classi RemoteFileInputStream e RemoteFileOutputStream. La classe RemoteFileInputStream deve implementare l’interfaccia RemoteFileInputStreamInterface mentre invece la classe RemoteFileOutputStream invece deve implementare l’interfaccia RemoteFileOutputStreamInterface. La semantica dei metodi è quella dei corrispettivi metodi delle classi FileInputStream e FileOutputStream. Il costruttore del RemoteFileOutputStream invece è leggermente diverso perchè prende come argomento il nome del file da creare, piuttosto che un descrittore di stream. Le due interfacce da implementare sono le seguenti:

```
public interface RemoteFileInputStreamInterface {

	public int read(byte[] buffer, int offset, int len) throws InternalErrorRemoteFileException;
	/**
	 * this closes the file opened for reading.
	 * @throws InternalErrorRemoteFileException in case of unrecoverable error while closing
	 */
```

```
    public void close() throws InternalErrorRemoteFileException;

}


public interface RemoteFileOutputStreamInterface {


    /**
     * writes len bytes from buffer, starting at offset to the file
     * @param buffer the bytes to be written
     * @param offset the offest to start taking bytes to write
     * @param len how many bytes have to be written
     * @throws InternalErrorRemoteFileException in case of unrecoverable error while writing
     */
    public void write(byte[] buffer, int offset, int len) throws InternalErrorRemoteFileException;

    /**
     * closes the handle.
     * @throws InternalErrorRemoteFileException in case of unrecoverable error while closing
     */
    public void close() throws InternalErrorRemoteFileException;

}
```

Gli oggetti di tipo RemoteFileInputStream deve prevedere un costruttore

```
   /**
     * constructor. This constructs the object used to access the remote file for reading.
     * @param name the name of the file to be opened for reading
     * @throws FileNotFoundException in case the file does not exist in the system
     * @throws AlreadyOpenRemoteFileException in case the file is already open for writing
     */
    public RemoteFileInputStream(String name) throws RemoteFileNotFoundException,
                                                     AlreadyOpenRemoteFileException;
    /**
     * this reads from the remote file a number of bytes into a buffer
     * @param buffer the destination buffer
     * @param offset the offset in the destination buffer where bytes have to be inserted
     * @param len the maximum amount of bytes to be read in the buffer
     * @return the number of bytes actually read in the buffer
     * @throws InternalErrorRemoteFileException in case of unrecoverable error while reading
     */
```

e gli oggetti di tipo RemoteFileOutputStream devono prevedere un costruttore tipo

```
    /**
     * constructor. If the file exists, it is truncated and rewritten or the data
     * are appended to the end of the existing file, depending on the append flag.
     * If the file already exists, and it is already open, then an exception is returned.
     * @param name The name of the file to be written
     * @param append true id the data has to be appended to an existing file, false
              if an existing file has to be truncated
     * @throws AlreadyOpenRemoteFileException if the file already exists and there are
               active readers or writers on that file
     * @throws CannotCreateRemoteFileException if the file cannot be created as expected
     */
    public RemoteFileOutputStream(String name, boolean append)
                throws AlreadyOpenRemoteFileException, CannotCreateRemoteFileException;
```

Le due classi RemoteFileInputStream e RemoteFileOutputStream ***devono*** rispettare i vincoli appena enunciati in termini di nomi, metodi e signature dei metodi. Le uniche modifiche permesse riguardano la possibilità di lancio di (altre) eccezioni da parte dei metodi.

Si dovrà altresì realizzare una classe SecureFileSystemManagement che permetta l'accesso allo stato interno del file system sicuro, implementando i metodi dell'interfaccia SecureFileSystemManagementInterface.

```
public interface SecureFileSystemManagementInterface {

	/**
	 * returns the number of files (not the number of copies) currently in the secure file system
	 * @return the number of files in the secure file system
	 */
	public int remoteFileNumber();
	/**
	 * used to know the current number of readers for a file
	 * @param filename the name of the file
	 * @return the current number of reader handles
	 */
	public int remoteFileReaders(String filename);
	/**
	 * used to know the current number of writers for a file
	 * @param filename the name of the file
	 * @return the current number of writer handles
	 */
	public int remoteFileWriters(String filename);
	/**
	 * used to know the overall number of readers active on the secure file system
	 * @return the number of active readers
	 */
	public int remoteFileReaders();
	/**
	 * used to know the overall number of writers active on the secure file system
	 * (each writer operates on the different file, according to the secure file
	 * system semantics)
	 * @return the number of active writers
	 */
	public int remoteFileWriters();
	/**
	 * used to collect information about all the file stored in the secure file system.
	 * Each entry in the String array is related to a single file.
	 * The following information has to be represented in the entry:
	 * <ul>
	 * <li> the name of the file (this must be the first information in the
	      string, separated from the rest of the information by a space or by a tab </li>
	 * <li> its current lenght </li>
	 * <li> the name of the Disco server hosting copies of the file </li>
	 * <li> the number of readers active on the file </li>
	 * <li> the number of writers active on the file </li>
	 * </ul>
	 * @return
	 */
	public String [] remoteLs();
	/**
	 * This call performs a complete check of the secure file system. For all the files
	 * in the system, a check is performed to ensure that at least K copies are present.
	 * In case this is not true, a recovery action has to be taken. We assume this operation
	 * is only performed in case there are no active writers, otherwise an
	 * exception is returned.
	 * @return true if the check is passed, false otherwise
```

```
     * @throws AlreadyOpenRemoteFileException in case there are writers active
     */
    public boolean remoteFsck() throws AlreadyOpenRemoteFileException;
    /**
     * terminates all the processes implementing the system, including FileSystem and Disk
     * processes. All the files stored into the system are left. It is up to the student to
     * decide if file copies have to be left in the file system or to be removed.
     *
     */
    public void shutdown();
    /**
     * terminates the processes as in the shutdown(). In addition, the files are removed from
     * local file system of the Disk processes and a (single) copy of the files is placed in
     * a subdirectory of the local /tmp
     * @param dirname the name of the /tmp subdirectory to be used to store the files
     */
    public void dump(String dirname);


}
```

Meccanismi di interazione

L'interazione fra le classi **RemoteFileInputStream** e **RemoteFileOutputStream** e il processo FileSystem *deve* avvenire utilizzando RMI. La localizzazione dell'istanza del processo FyleSystem da utilizzare per l'accesso al file system sicuro deve avvenire utilizzando multicast UDP. L'interazione fra processi FileSystem e processi Disco può avvenire secondo le modalità ritenute più opportune. Nella relazione di accompagnamento del progetto dovranno essere motivate le scelte del protocollo di trasporto e/o del protocollo applicativo utilizzato per questa comunicazione. Nella realizzione del progetto deve essere previsto l'uso di socket TCP/IP per almeno uno dei casi di interazione/comunicazione fra processi e librerie del file system sicuro.

Operazioni sul file system sicuro

Apertura di un file in output

L'apertura di un file in output puà avvenire solo quando non vi siano processi lettori o scrittori attivi su quel file. Nel caso ve ne siano, l'apertura (ovvero la dichiarazione di un **RemoteFileOutputStream**) deve terminare con un'eccezione di tipo **AlreadyOpenRemoteFileException**. Nel caso l'apertura del file vada a buon fine, il file system sicuro deve garantire che nessun altro processo potrà richiedere descrittori sul file aperto, nè in lettura nè in scrittura. L'apertura di un file in scrittura (sia in append che in modalità normale) già presente nel sistema, deve procedere attraverso la ridenominazione del file precendente in un file con lo stesso nome seguito dal suffisso "**.orig**". Qualora esistesse già nel file system sicuro un file con lo stesso nome e suffisso, questo sarà sovrascritto.

Durante la procedura di apertura del file, il server FileSystem dovrà provvedere a trovare **K** serventi Disco diversi e a contattarli perchè possano ospitare ognuno una copia del file che verrà scritto. (**K** è il parametro che determina a tempo di compilazione il grado di replicazione dei singoli file memorizzati nel file system sicuro). I server Disco provvederanno a memorizzare il file in una sottodirectory **RemoteFileOutputStream.HOMEDIRECTORY** della directory **/tmp** della macchina su cui sono in esecuzione.

Scrittura su un file

Una operazione di scrittura comporta, da parte del FileServer, la scrittura del buffer su tutte le K copie del file, mantenute da K processi Disco diversi.

Chiusura di un file aperto per la scrittura

La chiusura di un file aperto per la scrittura comporta la chiusura e la memorizzazione definitiva sul file system locale sui K processi Disco che implementano le K copie del file nel file system sicuro.

Apertura di un file per la lettura

L'apertura di un file per la lettura comporta la verifica che non vi siano processi che stanno scrivendo quel file. A seguito di una richiesta di questo tipo (creazione di un oggetto **RemoteFileInputStream**) il processo FileSystem contatta i K server Disco che possiedono una copia del file. Nel caso in cui non si riescano *ragionevolmente* a contattare K processi si provvede a delagare altri processi Disco affinchè il numero di copie del file in questione sia effettivamente K. Questo può richiedere il trasferimento dell'intero file da uno dei processi Disco rimasti attivi ai nuovi processi Disco coinvolti nella gestione del file.

Lettura da un file

La lettura di un blocco del file comporterà la lettura del blocco sui K server Disco interessati e la restituzione immediata di una copia del blocco letto al cliente *se e solo se* i tutti i *k* blocchi letti risultano uguali. In caso contrario, il server FileSystem deve procedere ad un controllo: se esiste una maggioranza di blocchi uguali, al cliente a restituito il valore "di maggioranza" del blocco e si deve contemporaneamente provvedere a far partire un'azione di recovery che preveda la sostituzione dei blocchi dicersi con il blocco che rappresenta "a maggioranza" fra i blocchi letti, il valore corretto. Se invece *tutti* i blocchi fossero diversi fra loro, deve essere restituita un'eccezione al cliente, senza alcun valore per il blocco.

Chiusura di un file aperto in lettura

La chiusura di un file aperto in lettura comporta la chiusura dei relativi file sui K server Disco che memorizzano le K copie.

Maggiori dettagli sulle varie operazioni si possono ricavare dai flowchart riportati in fondo al documento.

Modalità di consegna

La consegna del progetto avviene per email e consta di tre parti distinte:

1. il codice
2. una relazione
3. la documentazione JavaDoc

I tre oggetti devono essere allegati in un messaggio diretto al docente il cui subject *deve* essere (verranno filtrati in automatico, e dunque quelli che non rispettano il formato non saranno presi in considerazione):

Consegna LPRb.Appello.Numero Cognome.Nome Matricola

ad esempio

Consegna LPRb.Appello.1 Rossi.Mario 543210

La consegna può avvenire in un momento qualunque delle due settimane di appello. L'ultimo giorno di appello, o nei giorni immediatamente seguenti il docente provvederà a pubblicare la lista dei progetti consegnati e a fissare le date per le discussioni.

Codice

Il codice deve essere consegnato in formato sorgente (file zip, jar o tar.gz). Devono essere inclusi nel codice sia i file relativi all'implementazione che eventuali file (Java, script o testo) da usare per il test del corretto funzionamento del progetto. Il codice deve essere Java 1.5 compliant (questa è la versione di Java disponibile al momento su tutte le architetture normalmente utilizzate, e cioè Windows, Linux e Mac OS/X). In caso di mafunzionamento o contestazioni, le prove del progetto verranno effettuate sui fujih*.cli.di.unipi.it configurati in modo da utilizzare Java 1.5.

Relazione

La relazione non deve essere più lunga di 10 pagine, tutto compreso. Deve contenere una sezione che illustri tutte le scelte progettuali, a livello di astrazione piuttosto alto, una sezione che raccolga i dettagli relativi all'implementazione qualora si siano adottate scelte particolari, e una sezione contenente tutti i comandi e le azioni necessarie per realizzare un esperimento con il progetto così come consegnato (manuale d'uso). Qualora non fosse presente il manuale d'uso il progetto non verrà accettato.

Javadoc

In un file di tipo zip o tar.gz va consegnata tutta la documentazione JavaDoc generata dal codice sorgente.

Flowchart indicativi per le operazioni di apertura e lettura/scrittura