Sommario

[1. Server 2](#_Toc266434662)

[1.1 Caratteristiche principali del server 2](#_Toc266434663)

[1.2 Servizi offerti dal server 3](#_Toc266434664)

[1.2.1 Servizi offerti dal server ai server: Lista File 4](#_Toc266434665)

[1.2.2 Servizi offerti dal server ai server: Richiesta di commit 4](#_Toc266434666)

[1.2.3 Servizi offerti dal server ai server: Aggiorna File 5](#_Toc266434672)

[1.2.4 Servizi offerti dal server ai server: Copia File 5](#_Toc266434673)

[1.2.5 Servizi offerti dal server ai server: Uscita 6](#_Toc266434674)

[1.3 Implementazione del server 6](#_Toc266434675)

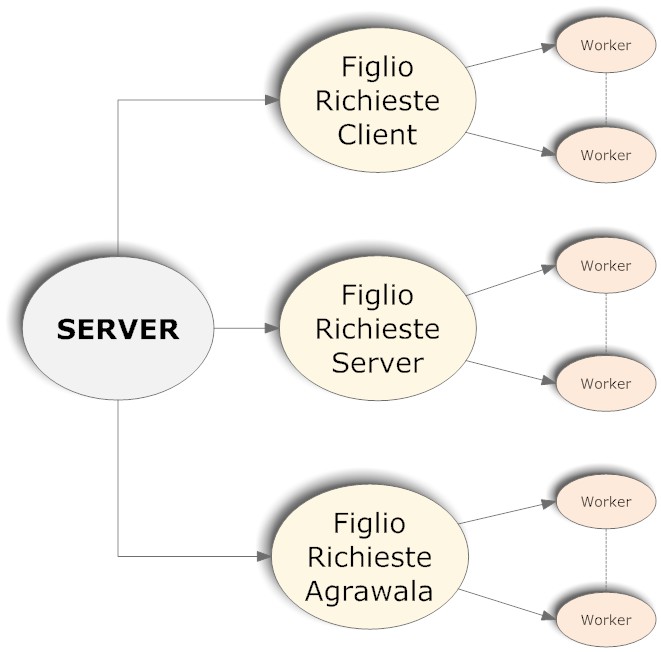
[1.3.1 Pacchetto Applicativo 6](#_Toc266434676)

[1.3.1 Trasferimento di file tra un server e un client o tra server e server 7](#_Toc266434677)

# Server

## Caratteristiche principali del server

Il Server da noi implementato è di tipo non bloccante. E’ in grado di gestire perciò più richieste provenienti da vari client o server diversi. Ciò avviene grazie all’utilizzo dei processi.  
All’avvio il server crea tre processi: Il primo processo è in attesa di richieste da parte di Client, il secondo è in attesa di richieste da parte dei server e un terzo processo è utilizzato per attuare il meccanismo di agrawala.



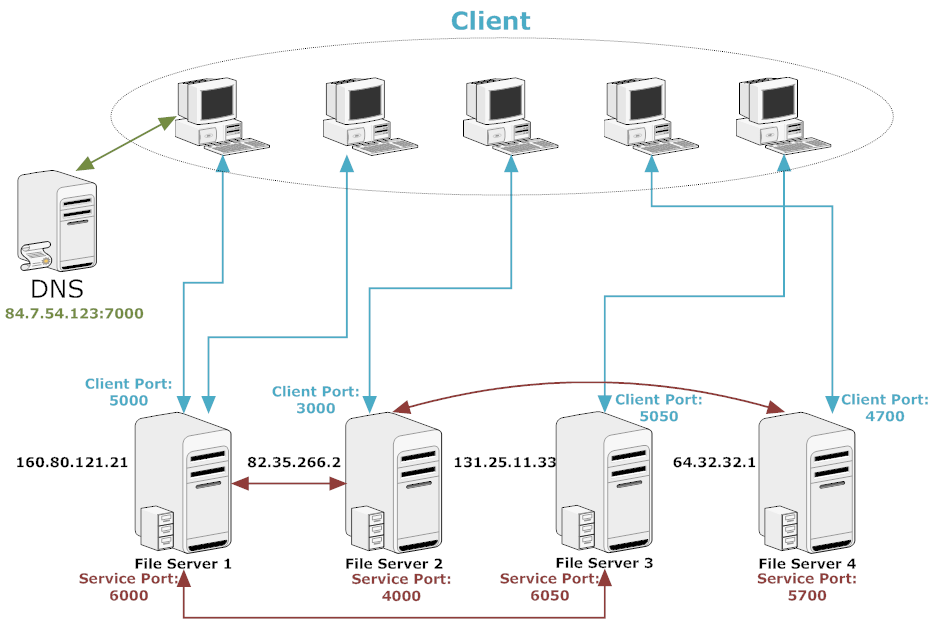
I processi in attesa di richieste da parte di client o server, creano ognuno un figlio nel momento in cui arriva una richiesta. Dopo aver creato un figlio, sarà esso che si occuperà di gestire la richiesta e il processo padre, che era in ascolto, torna ad essere in ascolto pronto a soddisfare nuovamente le richieste di altri client e a creare se necessario altri figli. Al processo figlio verrà passato il descrittore del socket creato dalla accept e il processo padre, prima di tornare in ascolto, chiude il socket creato dalla accept per risparmiare le risorse del server. Il figlio difatti, essendo un processo, ha una copia di questo socket e lavorerà tramite essa.   
Il numero massimo di figli creabili da entrambi i padri in attesa di richieste è 50. Questo per evitare che il server si sovraccarichi di richieste e non riesca a gestirle tutte.  
Una delle operazioni effettuate durante l’avvio del server è la sincronizzazione con gli altri server. Durante la fase di avvio verranno contattati gli altri server per ricevere i file presenti sul file system distribuito. In questo modo, sia che si tratti del primo avvio, sia che si tratti di un avvio a seguito di un crash, il server avrà la stessa versione dei file presenti sul file system distribuito.   
I file che saranno resi disponibili sul file system distribuito sono salvati in una cartella definibile tramite il file di configurazione. I file presenti in questa cartella sono uguali per tutti i server che si trovano in esecuzione e che fanno parte del file system distribuito.  
Altri file utilizzati dal server sono:

* Il file di configurazione, configurazioneServer.cfg, che contiene una serie di specifiche che permettono di avviare il server secondo una data configurazione.
* Un file di Log, che contiene tutte le operazioni che il server sta svolgendo durante la sua esecuzione.
* Vari file di PIPE utilizzati per implementare il meccanismo di Agrawala. Essi sono salvati nella cartella /tmp di Unix
* Altri file temporanei creati mentre il client sta effettuando l’operazione di scrittura file ma ancora non ha effettuato il commit

Per poter attuare il meccanismo di Agrawala e poter fare in modo che ogni processo del server possa conoscere la lista dei file aperti dal server in un dato istante, viene utilizzata la memoria condivisa. Nella memoria condivisa è presente un’array che contiene la lista dei file attualmente aperti dal server e in uso. L’utilizzo di questa lista è spiegato nei capitoli X.X e X.X  
Il server, riceve e invia gli aggiornamenti relativi ai file presenti nel proprio file system distribuito in modalità push. Nel momento in cui un client effettua una modifica su un file presente nel file system distribuito, essa verrà inviata agli altri server nel momento del commit. Grazie a questa scelta possiamo avere un grado di consistenza molto alto e garantire che i server abbiano la stessa versione del file quasi nello stesso istante.

## Servizi offerti dal server

Per poter realizzare un file system distribuito il server fornisce vari servizi sia ai client sia agli altri server facenti parte del file system distribuito. Durante l’esecuzione, il server attende le richieste su due porte diverse, una dedicata ai server e l’altra dedicata alle richieste provenienti dai client.



Le operazioni invocabili e rese disponibili al client sono:

* Lista File: Il server invia al client la lista dei file presenti sul file system distribuito.
* Leggi file: Il server invia al client un intero file presente sul file system distribuito.
* Scrivi file: Il server permette al client di effettuare una sequenza di operazioni di scrittura su di un file che sarà visibile all’interno del file system distribuito solo al termine della sequenza di operazioni e tramite l’operazione di commit
* Uscita: Il server termina la connessione con il client

Le operazioni invocabili e rese disponibili invece agli altri server sono:

* Lista File: Come nel caso del client, invia ad un altro server la lista dei file presenti sul file system distribuito
* Richiesta di commit: Questa operazione permette, ad un altro server, di sapere se può effettuare il commit in base all’algoritmo di agrawala.
* Aggiorna file: Permette al server aggiornare un file presente sul file system distribuito.
* Copia File: Invia un file ad un altro server. Il file è spedito per intero.
* Uscita: Permettere di chiudere la connessione con il server

Il funzionamento di ogni operazione è spiegato dettagliatamente nei paragrafi successivi.

La porta di ascolto per i server è calcolata prendendo il numero di porta che i client possono contattare e aggiungendo mille unità. Ad esempio, se il server è pronto a servire i client sulla porta 5000, la porta di ascolto per gli altri server, sarà 6000. In questo modo la configurazione del server sarà più semplice da realizzare e vedremo, nel capitolo X.X, che grazie a questa soluzione il DNS fornirà una sola porta di ascolto, quella dei client. La porta di servizio, se necessaria, potrà essere calcolata a tempo di runtime dal server.

### Servizi offerti dal server ai server: Lista File

Questa operazione permette di inviare la lista completa dei file presenti sul file system distribuito, ad altri server.  
Il figlio di servizio che ha preso in consegna la richiesta da parte di un altro server, per poter inviare la lista file, dovrà ricevere un pacchetto applicativo in cui il tipo operazione sia uguale a “lista file”. Dopo aver effettuato questo controllo, provvederà ad effettuare una scansione della cartella locale dove il server contiene i file presenti nel file system distribuito. Preparerà quindi un pacchetto applicativo con il tipo di operazione sempre di tipo “lista file” e all’interno, nella parte di messaggio, inserirà la lista dei file. Se la lista dei file supererà la dimensione di 500byte, verranno spediti più pacchetti applicativi.

### Servizi offerti dal server ai server: Richiesta di commit

Tramite questa operazione, un qualsiasi altro server richiede al server se può effettuare il commit in base alla precedenza assegnata tramite il meccanismo di Agrawala.  
In questo paragrafo, il server che riceve la richiesta e che deve fornire il servizio verrà chiamato server A mentre colui che effettua la richiesta verrà chiamato server B.   
Il figlio di servizio del server A che ha preso in consegna la richiesta da parte del server B controllerà che il tipo di operazione ricevuto sia “chiedo di fare commit”. Il pacchetto applicativo ricevuto conterrà inoltre anche il nome del file di cui il server B intende effettuare il commit e l’ID. Il server A procederà a controllare nella memoria condivisa, che il file non sia presente nell’array della lista dei file aperti. Le possibili situazioni che si possono verificare sono due:

1. Se il file non è presente nella lista dei file aperti, il server A invierà al server B un pacchetto applicativo con il tipo operazione uguale a “conferma per il commit” e nel messaggio inserirà “ok”. Se il file non è presente nell’array difatti, siamo nella situazione in cui il server A non ha nessun client che sta effettuando operazioni di scrittura su quel file e quindi può inviare senza problemi la conferma al server B.
2. Nel caso in cui invece, il file è presente nella lista dei file aperti, il server A deve controllare chi ha la precedenza per effettuare il commit. In questo caso la precedenza per il commit è data al server che ha l’ID più basso. Possiamo quindi distinguere due ulteriori situazioni possibili:  
   1. Il server A ha ID minore del server B: In questo caso non invierà nessuna conferma al server B che rimarrà in attesa di una risposta. La risposta verrà difatti inviata solo nel momento in cui il server A vedrà sparire dall’array della lista dei file, il file di cui il server B intende effettuare il commit. Il server A difatti, avendo ID minore, ha la precedenza sull’effettuare il commit rispetto al server B. Una volta effettuato il commit può sbloccare il server B e permettergli di effettuare il commit.
   2. Il server A ha ID maggiore del server B: In questo caso il server A spedirà subito una conferma al server B in quanto B ha la precedenza nell’effettuare il commit. Nel momento in cui A dovrà effettuare il suo commit, procederà ad attuare il meccanismo di Agrawala.

Si ricorda che, secondo l’algoritmo di Agrawala, il server B per poter effettuare il commit non deve ricevere la conferma solo dal server A ma da tutti i server facenti parte del file system distribuito.



### Servizi offerti dal server ai server: Aggiorna File

Questa operazione effettua l’aggiornamento di un file presente nella cartella del file system distribuito. Una volta che un server ha ricevuto la conferma per il commit difatti, provvederà a aggiornare il proprio file locale e spedirà l’aggiornamento agli altri server.  
In questo paragrafo il server che vuole propagare l’aggiornamento è chiamato server A mentre il server che riceve la richiesta di aggiornamento è chiamato server B.  
Il server A invierà perciò a tutti gli altri server, un pacchetto applicativo con tipo operazione uguale a “aggiorna file”, un ID transazione generato pseudocasualmente e, nel campo nomefile, il nome file che dovrà essere aggiornato. Il server A effettua l’aggiormento inviando al server B solamente i dati aggiornati e non tutto il contenuto del file. Questo onde evitare uno scambio di dati troppo elevato.  
Scendiamo ora nei dettagli. Una volta che il server B riceve una richiesta di aggiornamento file, creerà un file temporaneo il cui nome sarà uguale all’ID transazione. Il file temporaneo è creato nella cartella /tmp di Unix. Una volta creato il file temporaneo, il server B avvisa il server A che è pronto a ricevere il file con i dati da aggiornare. Il server A procederà perciò a inviare al server B il file. Una volta ricevuto il file per intero, il server B procederà ad “appendere” al file da aggiornare i nuovi dati. La scrittura del file è eseguita effettuando un lock sul file da aggiornare. Questo per evitare che sul server B ci sia un altro client che stia effettuando un’operazione di commit nello stesso istante sullo stesso file.

### Servizi offerti dal server ai server: Copia File

Questa operazione permette ad un server di inviare un file ad un altro server. In questo paragrafo, il server che effettua la richiesta di un file è il server A mentre colui che riceve la richiesta e che dovrà spedire il file è il server B. Il server A spedirà al server B un pacchetto applicativo, sulla porta di servizio, con il tipo operazione settato a “copia file”. All’interno del campo nomefile, inserirà il nome del file che vuole ricevere e nel campo IDtransazione, inserirà un ID generato pseudo casualmente. Il server B controllerà che il file richiesto dal server A esista e procederà ad inviare al server A un pacchetto applicativo con tipo operazione settata a “copia file, pronto”. In questo modo avvisa A che il file esiste e che B è pronto a inviarlo. Il server A risponderà con un pacchetto applicativo il cui tipo operazione è settata a “copia file, pronto a ricevere” che server a informare B che può cominciare a inviare il file ad A. Terminato l’invio del file, il server B si rimetterà in ascolto di una nuova eventuale operazione da parte di A. Nel caso in cui, A non sia interessato a effettuare una nuova operazione, la connessione sarà chiusa tramite l’operazione “Uscita”.  
Se il file richiesto da A non esiste, verrà inviato un pacchetto applicativo con tipo operazione “copia file, non trovato” in modo tale da informare A della non presenza del file. Il server B si rimetterà in ascolto di eventuali altre richieste da parte di A.

### Servizi offerti dal server ai server: Uscita

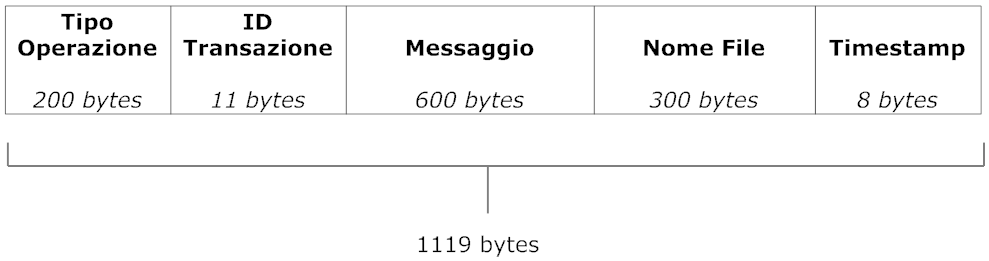
Questa operazione permette ad un server (A) di chiudere la connessione stabilita con il server (B). Il server A invierà un pacchetto applicativo con tipo operazione settato a “Uscita”. Il server B manderà un messaggio di conferma ad A con il tipo operazione settata a “arrivederci”. Dopodiché procederà a terminare il figlio che serviva le richieste provenienti da A. Il server A, ricevuto il messaggio di conferma chiusura, procederà anche lui a chiudere la connessione e ad effettuare la terminazione del figlio che connesso al server B.

## Implementazione del server

Verranno ora illustrate in che modo sono state implementate le funzionalità offerte dai vari server e quali sono state le varie problematiche affrontate e le scelte attuate per la risoluzione dei problemi riscontrati.

### Pacchetto Applicativo

Per permettere lo scambio di dati tra client e server e tra i vari server è stato creato un particolare tipo di pacchetto in modo da ridurre il numero di messaggi scambiati tra i vari client e server e avere con un solo pacchetto tutta una serie di informazioni utili all’esecuzione di una data operazione.  
Il pacchetto creato è così composto:



I campi sono utilizzati nel seguente modo:

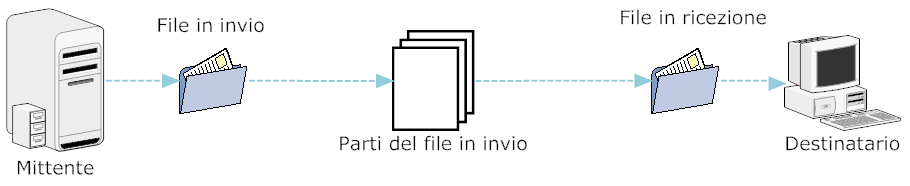
* *Tipo Operazione (200 byes):* Contiene una stringa con il tipo di operazione che dovrà essere effettuata dal server o dal client. Può contenere sia richieste che conferme da parte di entrambi le parte. E’ il primo campo che viene analizzato per verificare il tipo di operazione da attuare da parte del client o del server. E’ usato anche per controllare che il comportamento del client o del server non sia di tipo bizantino.
* *ID Transazione (11 bytes):* Contiene un ID transazione generato pseudocasualmente durante la copia di un file o l’esecuzione di una transazione da parte di un client. Nel caso in cui l’operazione richiesta non richiede un ID transazione (ad esempio, uscita), questo campo può essere vuoto.
* *Messaggio (500 bytes):* Contiene il messaggio complementare al tipo di operazione richiesta. E’ il corpo del pacchetto. A seconda del tipo di operazione in atto può contenere sequenze di byte (per la copia di file), o stringhe di testo. Se non utilizzato può essere vuoto.
* *Nome File (350 bytes):* Contiene l’eventuale nome del file che dovrà essere scritto o letto a seconda del tipo di operazione richiesta. Se non utilizzato può essere vuoto.
* *Timestamp (8 bytes):* Contiene un timestamp associato al tipo di operazione in atto. Il timestamp è generato numerando i pacchetti scambiati in modo sequenziale.

### Trasferimento di file tra un server e un client o tra server e server

Il trasferimento di file tra due parti connesse è realizzato tramite due funzioni, *riceviFile* e *spedisciFile*. Il mittente chiamerà la funzione *spedisciFile* mentre il destinatario chiamerà la funzione *riceviFile.* Per comodità, chiameremo con A la parte che si preoccupa di inviare il file e con B la parte che intende ricevere il file.

#### Spedisci File

La funzione *spedisciFile* prende in ingresso il socket connesso, l’ultimo pacchetto ricevuto e il descrittore del file da spedire. Il controllo sull’esistenza del file da spedire è effettuato prima di richiamare questa funzione. In caso il file non esista, la funzione *spedisciFile* non sarà richiamata e il mittente spedisce un pacchetto applicativo con all’interno un messaggio che avverte il destinatario che il file non è stato trovato.  
In caso in cui invece, il file esista, la funzione *spedisciFile* controllerà la dimensione del file da inviare e spedirà al destinatario un pacchetto applicativo con il tipo operazione settata a “leggi file, trovato”. Nel pacchetto applicativo verrà inserita anche la dimensione del file che dovrà spedire. In questo modo, il destinatario potrà effettuare tutta una serie di operazioni, che vedremo nella descrizione della funzione riceviFile, che gli permetterà di ricevere correttamente il file. Il mittente, spedito il pacchetto, calcolerà il numero di pacchetti applicativi necessari ad inviare il file e procederà ad inviarli. Il file è quindi suddiviso e inviato in tante parti. Per controllare che i pacchetti arrivino con lo stesso ordine al destinatario, ogni pacchetto contiene il numero di parte di file inviato. In questo modo a destinazione è possibile sapere se si sta ricevendo la parte corretta del file o se si è perso qualche pacchetto. Una volta terminato l’invio del file, la funzione ritorna il valore 1 se è riuscita ad inviare il file o 0 se il destinatario ha riscontrato problemi durante la ricezione del file.



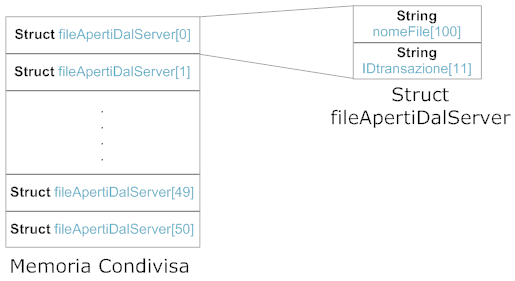
#### Ricevi File

La funzione *riceviFile* permette al destinatario di ricevere un file spedito da un mittente. Questa funzione prende in ingresso il socket connesso al mittente, il pacchetto ricevuto e il percorso, compreso di nome del file, dove andrà scritto il file. Questa funzione gestisce soltanto la ricezione del file, quindi il destinatario, prima di richiamarla, dovrà avvisare il mittente che intende ricevere un file. Per farlo, invierà un pacchetto applicativo con il tipo operazione settata a “leggi file” e il campo *nomefile* contenente il nome del file che intende ricevere. Se il mittente troverà il file, come visto in precedenza, spedirà un pacchetto applicativo con il tipo operazione settata a “leggi file, trovato”. Il destinatario, ricevuto questo pacchetto, provvederà a richiamare la funzione *riceviFile*.  
La funzione, procederà mettendosi in ascolto di un nuovo pacchetto che sarà spedito dal mittente, contenente la dimensione del file che si sta per ricevere. In questo modo, il destinatario può calcolare il numero di parti che dovrà ricevere e controllare se il file ricevuto è stato ricevuto interamente. Dopo aver calcolato il numero di parti, la funzione *riceviFile* prosegue cominciando a ricevere le parti del file. Durante la ricezione viene controllato che il tipo di operazione sia sempre uguale a “leggi file, trovato” e il numero del pacchetto ricevuto sia sequenziale. In questo modo si evitano comportamenti bizantini da parte del mittente e si evita che il destinatario si perda qualche parte del file.

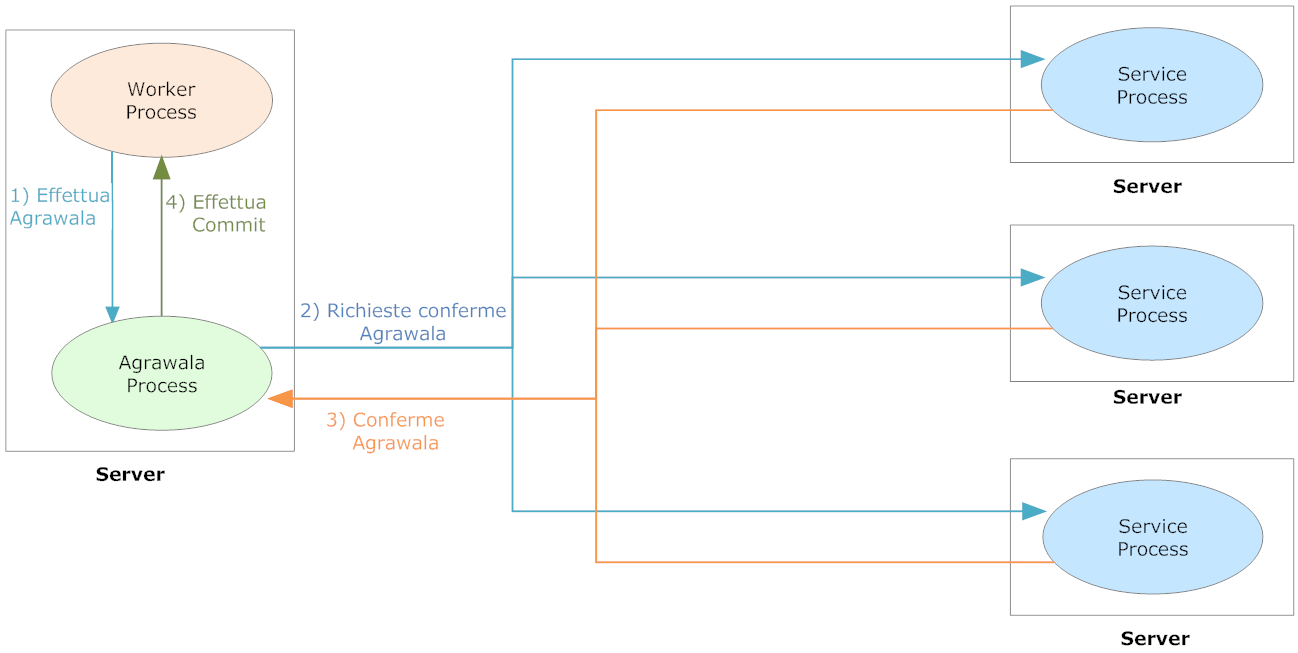
Entrambi le funzioni, spediscono il file inserendolo nel corpo del messaggio. Il file è suddiviso in tante parti da 600 bytes, ovvero la dimensione massima del corpo del messaggio. L’ultima parte del file può non essere grande quanto la dimensione massima del corpo del messaggio.  
Durante l’implementazione, prima di inserire un controllo sull’ordinamento dei pacchetti ricevuti, abbiamo riscontrato problemi di corruzione dei file. I file, giungevano difatti a destinazione ma risultavano corrotti anche se la dimensione del file ricevuto coincideva con quella del file inviato. Un caso particolare è stato quello di immagini Jpeg scambiate tra i due interlocutori. L’immagine ricevuta a destinazione veniva visualizzata con delle aberrazioni di colore o disturbi. Per quanto riguarda i file di testo invece, essendo di dimensione molto piccola, non si riscontravano problemi di lettura dei file.

### Comunicazione tra processi differenti e Agrawala

Per attuare l’algoritmo di Agrawala abbiamo dovuto scegliere e implementare un meccanismo che permetta a due processi diversi di comunicare tra di loro. Come spiegato nel capitolo 1.1 difatti, il meccanismo di Agrawala è eseguito da un processo diverso dai processi che si occupano di gestire le connessioni. In questo modo, il figlio che ha preso in carico una richiesta di Agrawala da parte di un client può rimanere in ascolto delle richieste del client mentre il processo di Agrawala si occuperà di contattare tutti gli altri server.  
In questo paragrafo, chiameremo con Agra, il figlio dedicato alle richieste di Agrawala, worker, il figlio che ha preso in carico una richiesta proveniente da un client, servent, il figlio che prende in carico le richieste provenienti da altri server sulla porta di servizio.  
I processi, possono comunicare tra di loro tramite varie modalità; attraverso lo scambio di messaggi con send e receive, attraverso una memoria condivisa o tramite pipe.  
Con la prima modalità ogni processo, per poter comunicare con un altro, deve essere a conoscenza del PID dell’altro processo. Questa modalità non è stata adottata in quanto, l’unico processo che può conoscere i PID degli altri processi generati a runtime è il padre. Agra inoltre, è in ascolto di richieste provenienti da molti processi e non da un solo processo.   
L’altra modalità prevede che i processi possano accedere ad un’area di memoria condivisa, che è letta da qualsiasi processo e può essere scritta da qualsiasi processo. Questa modalità è stata adottata, come vedremo in seguito, per fare in modo che un worker possa notificare ad Agra che deve cominciare a effettuare il meccanismo di Agrawala.  
Un’altra modalità prevede che i processi possano comunicare tra di loro tramite la pipe. La pipe è simile alla memoria condivisa. E’ un descrittore di file, presente sul file system, che può essere letto e scritto da due processi. Vedremo che la Pipe è utilizzata per fare in modo che Agrawala possa informare il worker che il commit può essere effettuato.  
Il server, all’avvio, prima di creare e avviare i vari processi, tra cui Agra, crea una memoria condivisa inserendo all’interno un array di 50 posizioni contenente una struct. La struct contiene al suo interno una stringa di 100 bytes e un’altra da 11 bytes per l’ID transazione.



L’array ha una dimensione di 50 posizioni in quanto, il numero massimo di client connessi al server è 50. Nella peggiore delle ipotesi quindi, il server si ritroverà ad avere 50 file aperti in attesa di avere una conferma di commit.  
Dopo aver creato la memoria condivisa, il server procede a creare il figlio Agra che si occuperà di gestire Agrawala. Agra, procederà a preparare le strutture per leggere la memoria condivisa. Dopodichè contatta il DNS per avere gli indirizzi degli altri server che sono presenti nel sistema e che deve contattare in caso di richieste di Agrawala da attuare. Dopo aver ottenuto gli IP dei server, Agra calcola qual è la porta di servizio su cui può contattare i server, aggiungendo mille unità alla porta comunicata dal DNS, questo in quanto il DNS gli fornisce le porte che i client devono contattare per avere i servizi dal server. Agrawala invece deve contattare il server sulla porta di servizio, che è quella su cui il server è in ascolto in attesa di richieste provenienti da altri server.  
Dopo aver effettuato il calcolo entra in un ciclo in cui controlla la presenza di una posizione non vuota all’interno della memoria condivisa. Infatti, quando un worker vuole effettuare Agrawala, per avvisare Agra deve inserire all’interno della memoria condivisa, nella prima posizione libera, il nome del file di cui intende effettuare il commit e l’ID transazione. Agra, non appena si ritrova una posizione non vuota, procede a contattare gli altri server con un pacchetto applicativo il cui campo tipo operazione è settato a “chiedo di fare commit”, nel corpo del messaggio il proprio ID e nel campo nomefile, il nome del file di cui il client intende fare il commit.



Il server che riceve sulla porta di servizio la richiesta di commit, andrà a controllare nella propria memoria condivisa la presenza del file. Se il nome del file non è presente nella memoria condivisa, siamo nel caso in cui il server contattato non ha nessun client che sta utilizzando il file richiesto dall’altro server e quindi, invia subito un pacchetto applicativo con tipo operazione uguale a “conferma per il commit” al server richiedente.  
Nel caso in cui invece, il nome del file è presente nell’array condiviso, il server che a ricevuti la richiesta controlla l’ID del server richiedente con il proprio. Se il proprio ID è minore, ha la precedenza nel commit e quindi, prima di inviare la conferma al server richiedente, effettuerà lui il commit e poi spedirà la conferma. Altrimenti, spedirà subito la conferma, e nel momento in cui lui dovrà effettuare il commit, avvierà una nuova procedura di Agrawala.  
Ritornando al server richiedente, Agra potrà effettuare il commit una volta che ha ricevuto le conferme da tutti i server comunicatigli dal DNS. Nel caso in cui un altro server stia effettuando il commit dello stesso file e ha la precedenza rispetto a lui, Agra rimarrà bloccato fino a che l’altro server non avrà terminato di effettuare il commit.  
Una volta ricevute tutte le conferme da tutti i server, Agra per avvisare il worker che ha preso in consegna la richiesta di commit, scriverà nella pipe il messaggio di conferma. La pipe è creata dal worker prima di scrivere nella memoria condivisa il file di cui si intende effettuare il commit. Per fare in modo che solo i due processi leggano la stessa pipe, essa ha come nome l’ID transazione utilizzato dal worker per scambiare dati con il client. Agra è a conoscenza dell’ID transazione grazie alla struct presente nella memoria condivisa. La struct contiene infatti, oltre al nome del file di cui si intende fare il commit, anche l’ID transazione associato ad esso e può quindi facilmente sapere quale sarà il file di pipe che dovrà aggiornare per avvisare il worker.  
Il worker, dopo che ha scritto nella memoria condivisa, si trova in un ciclo in cui controlla in continuazione il file di pipe per avere la conferma da Agra. Dopo aver ricevuto la conferma, il worker procede a scrivere il file nel file system del server e invierà l’aggiornamento agli altri server. Quando tutti i server hanno ricevuto l’aggiornamento, procede ad inviare una conferma al client.

### Spedizione degli aggiornamenti agli altri server

Dopo che il worker ha ricevuto la conferma di via libera per il commit, dopo aver effettuato la scrittura del file in locale, prima di confermare al client il buon esito dell’operazione deve spedire l’aggiornamento agli altri server in modo tale da avere preservare la consistenza del file system distribuito. L’operazione di aggiornamento è effettuata tramite la funzione *spedisciAggiornamentiAiServer* che prende in ingresso il descrittore del file contenente gli aggiornamenti da inviare, il nome del file da aggiornare, l’id del server che vuole effettuare l’aggiornamento e l’id transazione. La funzione procede nel richiedere al DNS gli indirizzi dei server facenti parte del file system distribuito. Anche in questo caso, la porta che utilizzerà per contattare i server sarà quella di servizio, quindi anche in questo caso, dopo aver ricevuto la porta dal DNS, calcoleremo quella di servizio aggiungendo mille unità. Dopo aver terminato il calcolo, la funzione comincia a spedire gli aggiornamenti ad un server alla volta. Per poter spedire un aggiornamento, il server crea un pacchetto applicativo con il tipo operazione settato a “aggiorna file”, il campo nomefile contenente il nome del file di cui si deve effettuare l’aggiornamento, il campo id transazione con l’ID transazione creato in precedenza dal worker. Il figlio di servizio del server contattato, risponderà con un pacchetto applicativo contenente all’interno il tipo operazione settato a “aggiorna file, pronto”. Da questo punto in poi, l’aggiornamento è inviato secondo le modalità descritte nel capitolo 1.3.1.  
Dal lato del server che ha preso in consegna la richiesta di servizio e che ha ricevuto il pacchetto applicativo con tipo operazione settato a “aggiorna file” , dopo aver ricevuto il file, provvederà ad unire gli aggiornamento ricevuti con il contenuto del file già presente nel proprio file system. Prima di effettuare l’unione con gli aggiornamenti ricevuti, il server effettua un lock del file che intende aggiornare. In questo modo si evita che un altro processo, che ha ricevuto le conferme per il commit, scrivi lo stesso file. E’ così preservata l’integrità del file e l’ordinamento delle scritture sul file del file system distribuito.

### Richiesta degli IP al DNS

Nei paragrafi 1.3.2 e 1.3.3 abbiamo illustrato come, il figlio di servizio e di Agrawala, per ottenere gli indirizzi degli altri server facenti parte del file system distribuito, procedono a contattare il server DNS. Illustreremo ora come viene contattato il server DNS e quali sono le operazioni che vengono effettuate dopo aver ricevuto la lista.  
La richiesta degli ip da parte dei processi è effettuata tramite la funzione *chiediTuttiGliIPalDNS* che prende in ingresso un array di struct di tipo sockaddr\_in dove verranno salvati gli indirizzi ricevuti dal DNS, l’IP e la porta del DNS da contattare e l’ID del server che sta effettuando la richiesta. La funzione procede a allocare le strutture necessarie a ricevere e a processare gli indirizzi DNS ricevuti dal server DNS. Successivamente procederà ad inviare un pacchetto applicativo con il tipo operazione settato a “indirizzi server” al server DNS. Il server DNS, ricevuta la richiesta, procederà a leggere un file contentente gli indirizzi, le porte e gli ID dei server attivi e li serializzerà all’interno di una stringa. La stringa verrà inserita all’interno del campo messaggio del pacchetto applicativo e verrà spedita al server richiedente gli indirizzi. Il tipo operazione settato è “indirizzi server”. Il server, ricevuta la risposta, controllerà che il tipo operazione ricevuta dal DNS sia “indirizzi server”, questo per evitare comportamenti bizantini da parte del DNS. Se il tipo di operazione è quello desiderato, procederà nel leggere il campo messaggio del pacchetto applicativo, che conterrà gli indirizzi IP serializzati. Effettuerà dunque una deserializzazione salvando IP e porta in una stringa e l’ID in una variabile di tipo int. Dopo questo procedimento, procederà a salvare nella struct sockaddr\_in ricevuta come parametro di ingresso, gli indirizzi IP e le porte dei server attivi. Durante la fase di salvataggio controllerà gli ID dei server corrispondenti agli IP e scarterà quello corrispondente al proprio ID. Questo perché al server interessa conoscere tutti gli IP dei server presenti, escluso il proprio, perché non è interessato a contattare se stesso su un’altra porta. Durante le fasi di debug ci è capitato di tralasciare questo dettaglio e l’effetto ottenuto è stato un comportamento anomalo del server durante le fasi di Agrawala o di aggiornamento dei file. In particolare, durante Agrawala, il server rimaneva appeso in attesa di risposte da lui stesso. Difatti, quando veniva effettuata la richiesta a lui stesso, il processo di servizio andava a controllare l’array dei file condivisi, in cui risultava che il file richiesto da lui stesso era già in uso. Si otteneva così una situazione di deadlock in quanto entrambi i figli, sia quello di Agrawala, sia quello di servizio, rimanevano in attesa che l’altro inviare una conferma per sbloccare la situazione.