# Testing delle Failure

Vediamo ora il comportamento del server e del client nel caso in cui siano presenti delle failure.

## Failstop

Il client prova a contattare un server che ha subito un crash. Il client ha un meccanismo che gli permette di capire se il client che sta per contattare è attivo o no. Dopo aver inoltrato una richiesta di connessione il client difatti, attende un timeout, scaduto il quale procederà a contattare un altro server. Nel caso in cui l’indirizzo IP e la porta del server che si sta per contattare risultino irraggiungibili si procede a contattare un nuovo server.

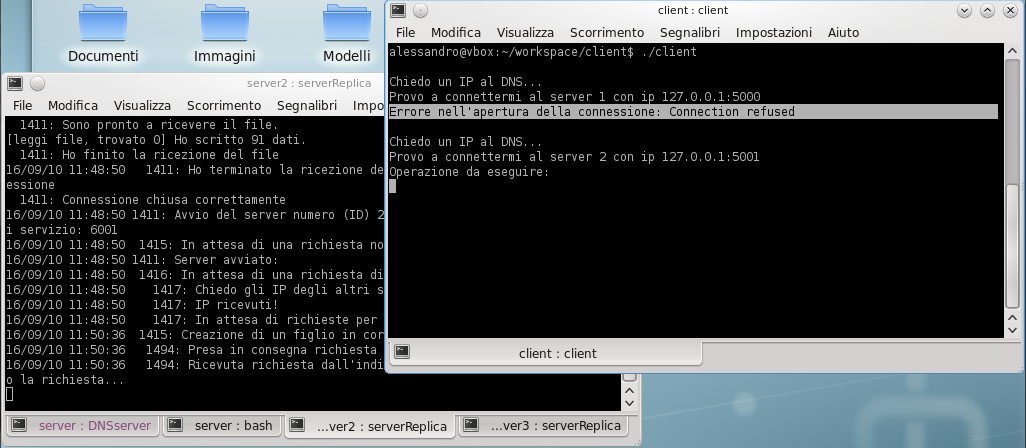


Figura : Failure del client

Il client, prima di contattare il nuovo server, procede a contattare il DNS. Difatti, ricordiamo che il DNS fornisce al client solo un indirizzo IP per volta, secondo un algoritmo di rotazione di tipo round-robin.  
Il timeout è assegnato al socket tramite la chiamata di sistema *setsockopt().* In particolare, è stata usata l’opzione SO\_RCVTIME0. Se il socket non riesce a contattare il server o se il timeout è scaduto, il socket viene chiuso e la variabile errno è settata a 111 e 11 rispettivamente.

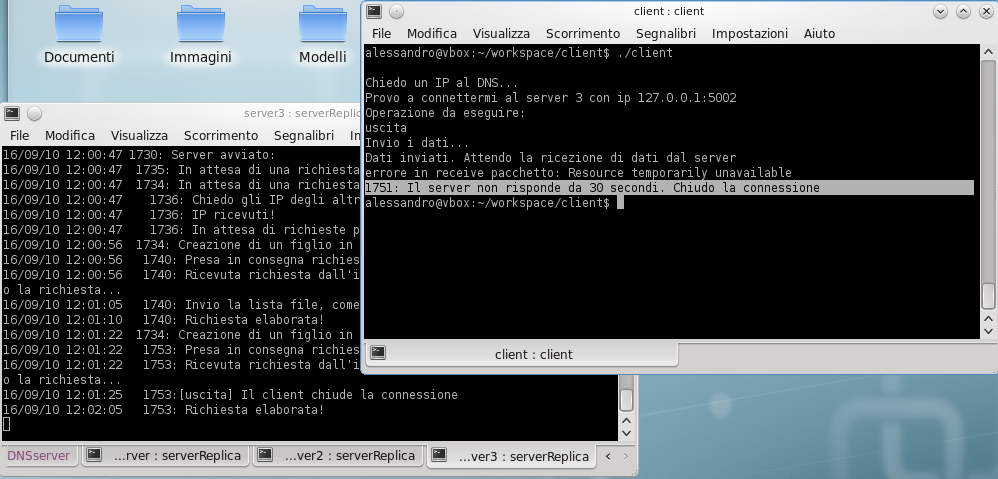


Figura : Crash del server. Timeout scaduto

Grazie a questa scelta, il timeout di attesa vale anche durante tutto il tempo in cui il client è connesso al server. Per cui, se il server subisce un crash durante la connessione con il client, il client chiude a sua volta la connessione e stampa a video un messaggio di errore. Questo tipo di failure è stata simulata inserendo una sleep > di 40 secondi all’operazione di uscita del server. In questo modo, il server attende 40 secondi prima di rispondere, un tempo molto maggiore rispetto al tempo di attesa di risposta da parte del client.

## Guasti bizantini

Vedremo ora come il client si comporta in caso di guasti bizantini da parte del server.  
Abbiamo visto nel capitolo X.X come durante ogni operazione svolta sia dal client che dal server, il pacchetto applicativo scambiato tra gli interlocutori contiene, nel campo tipo-operazione, l’operazione in atto. Il client e il server controllano difatti ad ogni scambio di messaggi, il tipo di operazione, in modo tale da tutelare i tipi di guasti di tipo bizantino. Difatti, se ad esempio, il server sta inviando al client la lista dei file, ci si aspetta che ogni pacchetto inviato dal server contenga l’operazione “lista file”. Il client controlla ad ogni pacchetto ricevuto che il tipo operazione corrisponda, e se ciò non accade interrompe la sua esecuzione, stampa a video un messaggio di errore e termina la propria esecuzione. Lato server invece, il server, passato un timeout, provvederà a chiudere la connessione e a terminare il processo figlio che stava gestendo la richiesta del client.  
I guasti bizantini sono anche evitati grazie all’utilizzo di un IDtransazione per le operazioni ritenute di maggiore importanza. Durante la copia di un file tra server, o la scrittura di un file da un client a un server, viene utilizzato una stringa pseudo-casuale di 10 caratteri chiamata IDtransazione. Quando un client vuole scrivere un file ad un server, il server provvede a assegnare al client un IDtransazione. L’IDtransazione è controllato sia lato server, sia lato client e ha validità fino a che non viene effettuato il commit della scrittura del file. Durante la sua validità, sia client che server, controllano che l’IDtransazione sia sempre lo stesso.

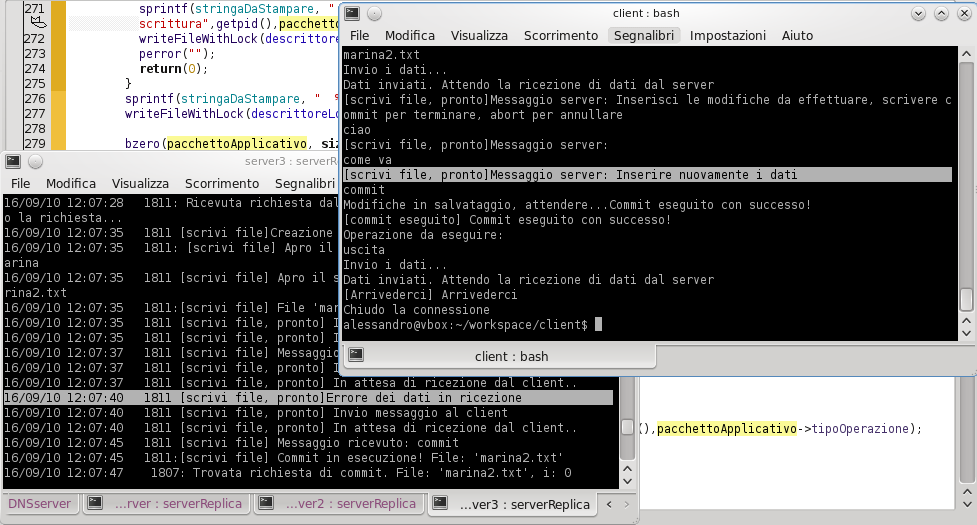


Figura : Failure bizantino con IDtransazione modificato

Nel caso in cui *l’IDtransazione* non corrisponda, il server prova a farsi rispedire dal client i dati, in modo tale da non annullare definitivamente la transazione. Nella figura possiamo vedere il caso in cui il server riceva un *IDtransazione* diverso da quello specificato in partenza. Questo tipo di failure è stata simulata inserendo nel server, durante la ricezione del pacchetto applicativo, una stringa diversa da quella *dell’IDtransazione*.

## Omissioni

Nel caso di omissioni, client e server hanno già stabilito una connessione e si stanno scambiando dei messaggi. Non siamo quindi nel caso in cui il client manda una sola richiesta e il server risponde. Abbiamo già visto nel paragrafo precedente, come l’omissione di una risposta, in questo caso, è risolta con il timeout.  
Durante lo scambio di messaggi, client e server, oltre a controllare che i campi *tipo operazione* e *IDtransazione* coincidano sempre con quelli aspettati, controllano anche il campo *timestamp*. In questo campo difatti, durante lo scambio di dati, viene salvato, sia dal client che dal server, il numero sequenziale del pacchetto inviato. In ricezione, entrambi quindi controlleranno che il numero di messaggio ricevuto sia maggiore di una unità rispetto al precedente. Se questa condizione non è verificata siamo nel caso in cui la rete ha perso un messaggio o ha scambiato l’ordine dei messaggi. Durante le fasi di implementazione, uno dei bug di funzionamento del client e del server, era dovuto proprio all’ordine dei messaggi. Durante la fase di scambio file difatti, il file ricevuto dal client risultava corrotto ma con la dimensione file uguale rispetto a quella inviata dal server.