Concept paper for Monitas

from Stefan Hübner
and the Monitas Team[*]

$Revision: 1.3 $, 2002-08-17

Abstract

This is a proposal how to design Monitas before actually start the coding. Collecting the ideas here will enforce to think about the structure before it is realized in code that cannot be modified later. With a little bit of luck this allows us to enhance Monitas' functionality quite easy in the future. And maybe we get a good user manual as side effect for free.

Revision 1.x of this *concept* paper is not describing the final version of the *software* that will be realized. It is loosely based on the existing alpha version 0.1 of Monitas and shall be a step by step approach towards Monitas version 0.99. If improvements to Monitas v0.x and this document will be in sync, we will have a good documentation of the internal system structure for Monitas v1.0 (and then this will be called Revision 2.0 of this document). So in general the major revision number of the concept paper is 1 higher than of the software, the minor revision numbers are both increasing but there is no connection between the values.

Contents

1 Basics **3**

2 Requirements for Monitas **3**

2.1 Start a backup from client 3

2.2 Start a backup from the server 4

2.3 Restore a backup from client 4

2.4 Restore a backup from the server 4

2.5 Compare backup from client 4

2.6 Compare from the server 5

3 System structure **5**

[*]Editor of this document is Stefan Hübner <shuebner.mondo@gmx.de>. Please send comments, extensions and proposals to him.

4 Message Flows **6**

4.1 Flows between Server and Client (*IFcs* and *IFsc*) 6

4.1.1 Connection Establishment 6

4.1.2 Backup and Restore Process 6

4.1.3 Connection Termination 6

4.1.4 List backups 6

4.1.5 List content of a backup 11

4.2 Message Flows between Client and (Graphical)User-Interface (*IFcg*) 11

4.2.1 (Re-)Connect to a server 12

4.2.2 Start a Backup 12

4.2.3 Start a Restore 12

4.2.4 Get list of previous, accessible Backups 12

4.2.5 Get content (files) of a specified Backup 14

4.2.6 Start a Verify 14

4.2.7 Start a Compare 14

4.2.8 Terminate Client 14

4.3 Message Flows between Server and (Graphical)User-Interface (*IFsg*) 15

4.4 Message Flows between Server and its Child (*IFss* and *IFsm*) . . 15

5 Interfaces **16**

5.1 Interface *Im* to Mondo 16

5.1.1 *Im* in Server 16

5.1.2 *Im* in Client 16

5.2 Interface *In* between Client and Server 16

5.3 Interface *Ipc*, Client's pipe to the GUI 16

5.4 Interface *Ips*, Server's pipe to the GUI 17

5.5 Interface *Is*, IPC of server instances 17

6 Commands at the GUI interfaces **17**

6.1 Message Structure 17

6.2 Implementation Detail 19

6.3 Defined Commands 19

7 Open Issues **19**

7.1 Possible Extensions 20

1 Basics

Monitas is an extension to mondo/mindi to deal with remote backups between TCP/IP connected PCs. Monitas consists of two logical parts: a *"client"* that resides on the PC where the files (to backup/restore) are and a *"server"* on another (or the same) PC where the backup is stored.

The backup may be written to/read from a CD, to a file on a hard-disk (or later to a revision controlled database).

Both together, server and client, will handle the backup and restore process in the background. Each is controlled via a pipe to send commands and to get status information back (error messages, progress information). Though it is possible to trigger the backup/restore process with these interfaces, they are mainly designed for graphical (or text based) front-ends which can dock there to control the process and interact with the user. These front-ends aren't described here, but the syntax and semantics of the interfaces.

We use a 1:n relation between the server and several clients on different PCs. That means there exists only one central server that serves all the clients out there.

Even if this document distinguishes between *server* and *client* and the different roles they play for the backup process, it is possible that both parts use the same executable as both parts have much code in common. Like *gzip*, which is compressing when executed as `gzip` and decompressing when executed as `gunzip`, Monitas will run as server when executed as `monitas_server` and will run as client when started as `monitas_client`.

2 Requirements for Monitas

This chapter contains the basic requirements for the client and the server part of Monitas.

2.1 Start a backup from client

Necessary input:

1. files to backup
2. type of the backup (CD, ISO-file, ...)
3. name of the backup (if not CD)
4. compression (method, client or server-side)

2.2 Start a backup from the server

Necessary input:

1. client to address (IP address or name)
2. files to backup
3. type of the backup (CD, ISO-file, ...)
4. name of the backup (if not CD)
5. compression (method, client or server-side)

2.3 Restore a backup from client

Necessary input:

1. files to restore
2. name/location of the backup (CD, ISO-file, ...)

2.4 Restore a backup from the server

Necessary input:

1. client to address (IP address or name)
2. files to restore
3. name/location of the backup (CD, ISO-file, ...)

2.5 Compare backup from client

There are 2 reasons for a compares
a) to guarantee the correct backup ("*verify*")
b) to find modifications of files since the last backup ("*compare*")

In case a) we must do a bit-compare between original files and their (decompressed) backup, in case b) it's sufficient to generate hash values of every original file and its backuped counterpart and compare the hashes (less network traffic).

To distinguish the different intentions, we call the comparison a) *Verify* as it's normally started directly after a backup, and call the case b) *Compare* as it is triggered to recognize modifications (perhaps to generate an incremental backup).

Necessary input:

1. name of the backup
2. mode of compare [a) or b)]
3. files to compare [mode b) only; mode a) will always compare all files in the backup]

2.6 Compare from the server

Necessary input:

1. client to address (IP address or name)
2. name of the backup
3. mode of compare [a) or b), the modes were described in previous subchapter]
4. files to compare [mode b) only; mode a) always compares all files in the backup]

3 System structure

Figure 3.1 gives an overview about the general structure of Monitas. Monitas' functionality is mainly split into 2 independent parts: a *client* component on the PC where files are backuped or restored, and a *server* component to write the backup on an external medium or read previous backups from an external medium.

Both parts base on mondo/mindi for accessing files, doing (de)compression, creating ISO-files, writing them to CD/DVD, ...

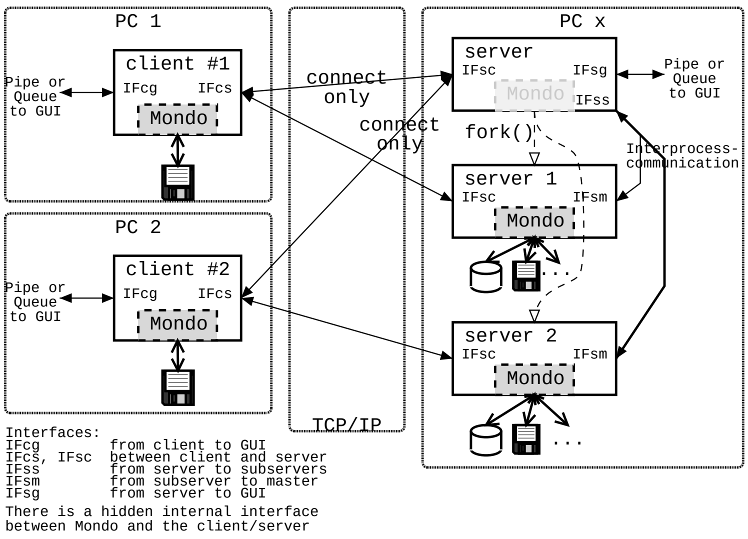


Figure 3.1: System structure

At startup, a client connects to the server process to establish the connection: The client uses a predefined ("well known") port where the server is listening. When receiving a message at this port the server duplicates itself by using the system call `fork()` or `clone()`. The child process will serve the connecting client on a new allocated port and the parent process will continue waiting for further clients.

By that mechanism we need only one predefined port number and only on the server PC. Details in the next chapter.

4 Message Flows

This chapter describes the information flow between Monitas' structural parts. The flows are designed to fulfill the requirements of chapter 2.

4.1 Flows between Server and Client (*IFcs* and *IFsc*)

4.1.1 Connection Establishment

Before any backup/restore can start, *client* and *server* must introduce each other. This message flow between (each) client process and the server (parent) process only takes place when a client process is started. The client calls the server to tell him "Here am I" whereupon the server is doubling itself via the `fork()` system call. The new created child of the server process will serve the new connected client from now on, while the parent process of the server continues waiting for other clients to connect. The connecting procedure is shown in Figure 4.1.

4.1.2 Backup and Restore Process

The backup and restore procedures are handled between the client and its corresponding child of the server process. Both, backup and restore may be triggered from server or from client side. The message flow between server and client for the backup process is shown in Figure 4.2.

The restore procedure shown in Figure 4.3 is very similar. It may be triggered either from server or from client side, too.

4.1.3 Connection Termination

When a client has done its job or some errors occurred (or maybe the server wants to stop running or ...) the connection can be shutdown in the way that Figure 4.4 shows.

4.1.4 List backups

Before restoring files, a client can inquire the server which backups are available. Since there are several locations where the server may store a backup (on CD, in local file(s), (in the future: in a database,) ...) the client can use the message flow defined in Figure 4.5 to get a list of all (for this client) accessible backups.

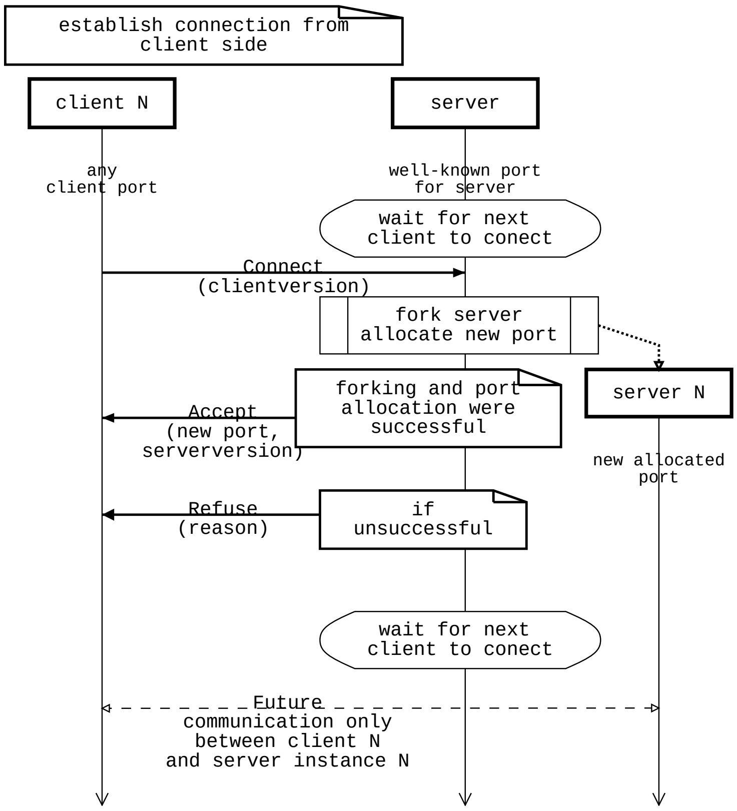

establish connection from client side
client N
server
any client port
well-known port for server
wait for next client to conect
Connect (clientversion)
fork server allocate new port
server N
forking and port allocation were successful
Accept (new port, serverversion)
new allocated port
Refuse (reason)
if unsuccessful
wait for next client to conect
Future communication only between client N and server instance N

Figure 4.1: A client connects to the server process

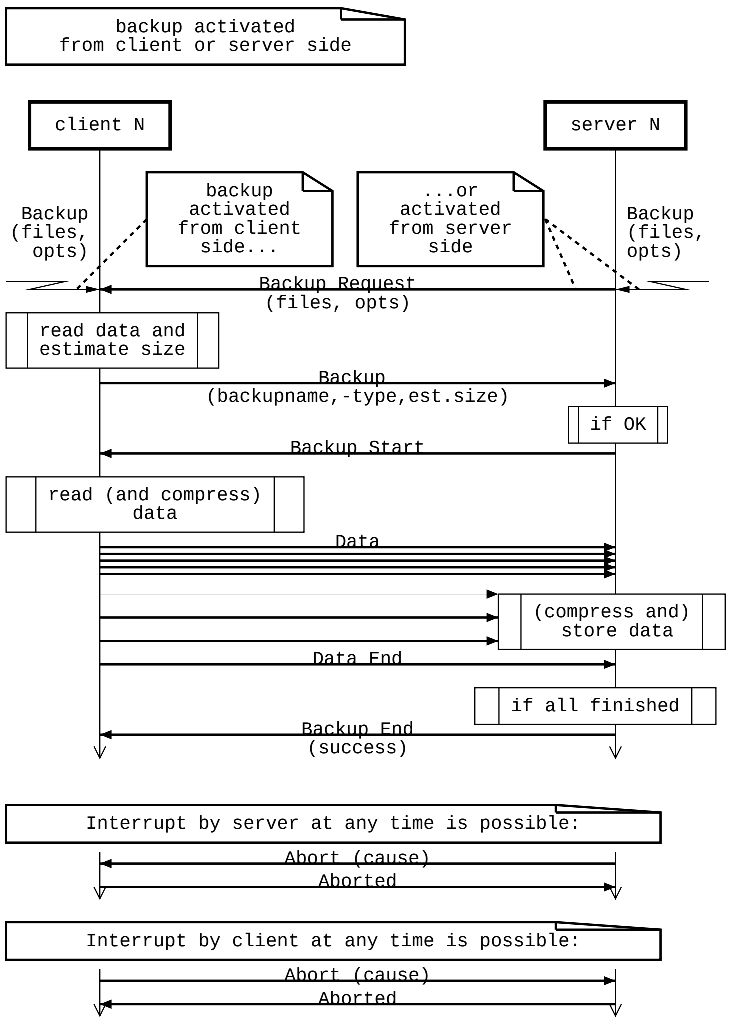
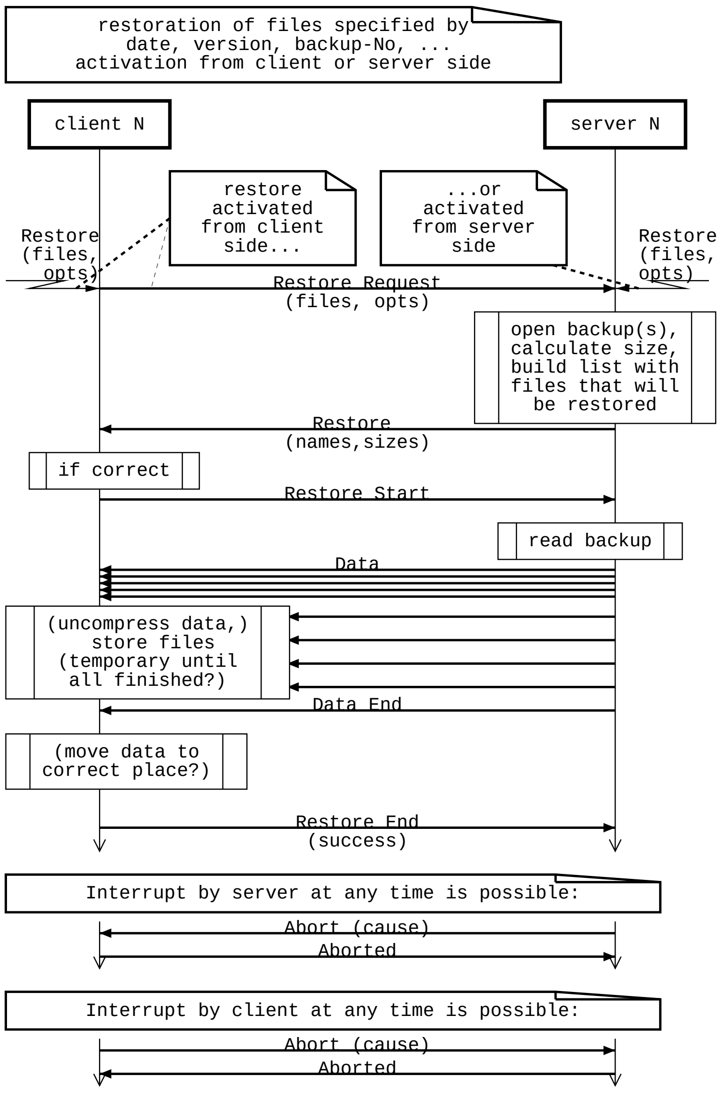
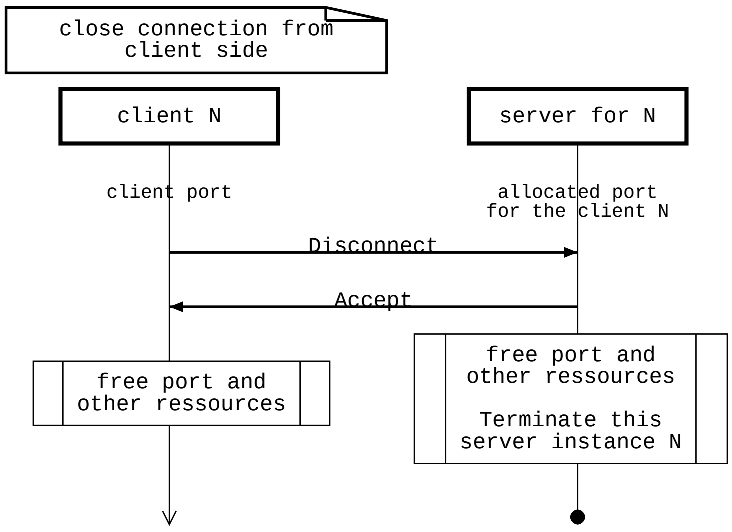

backup activated
from client or server side
client N
server N
backup activated from client side...
...or activated from server side
Backup (files, opts)
Backup (files, opts)
Backup Request (files, opts)
read data and estimate size
Backup (backupname,-type,est.size)
if OK
Backup Start
read (and compress) data
Data
(compress and) store data
Data End
if all finished
Backup End (success)
Interrupt by server at any time is possible:
Abort (cause)
Aborted
Interrupt by client at any time is possible:
Abort (cause)
Aborted

Figure 4.2: Message flow for Backup



restoration of files specified by date, version, backup-No, ... activation from client or server side
client N
server N
restore activated from client side...
...or activated from server side
Restore (files, opts)
Restore (files, opts)
Restore Request (files, opts)
open backup(s), calculate size, build list with files that will be restored
Restore (names,sizes)
if correct
Restore Start
read backup
Data
(uncompress data,) store files (temporary until all finished?)
Data End
(move data to correct place?)
Restore End (success)
Interrupt by server at any time is possible:
Abort (cause)
Aborted
Interrupt by client at any time is possible:
Abort (cause)
Aborted

Figure 4.3: Message flow for Restore


close connection from client side
client N
server for N
client port
allocated port for the client N
Disconnect
Accept
free port and other ressources
free port and other ressources Terminate this server instance N

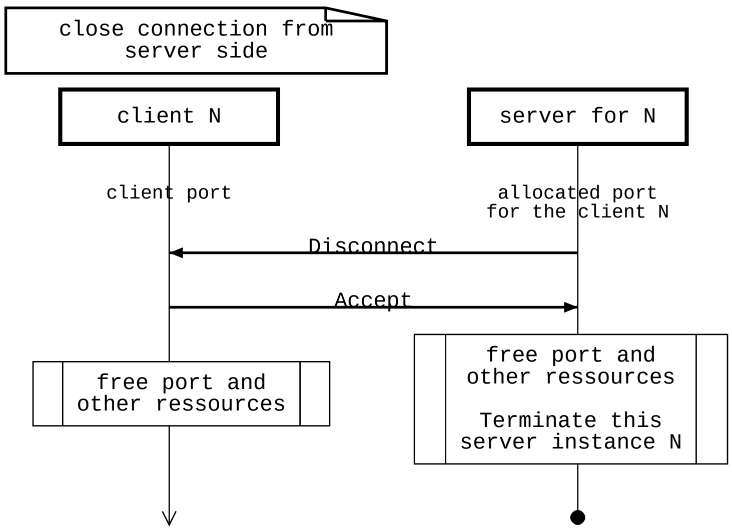

close connection from server side
client N
server for N
client port
allocated port for the client N
Disconnect
Accept
free port and other ressources
free port and other ressources Terminate this server instance N

Figure 4.4: Terminating the connection between server and client

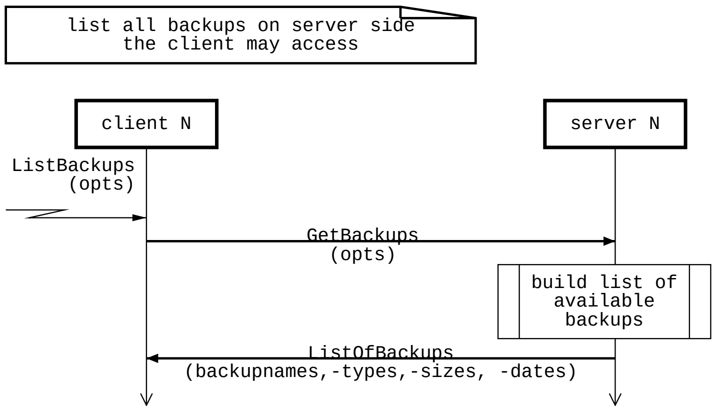

list all backups on server side the client may access
client N
server N
ListBackups (opts)
GetBackups (opts)
build list of available backups
ListOfBackups (backupnames,-types,-sizes, -dates)

Figure 4.5: Get list of accessible backups from server

4.1.5 List content of a backup

When doing a *nuke restore* it's sufficient to address a total backup. But in all other cases you want to know which files are in the backup, what their sizes, modification dates are and what other info is available. To inquire the content of a specific backup file, the client uses the flows in Figure 4.6.

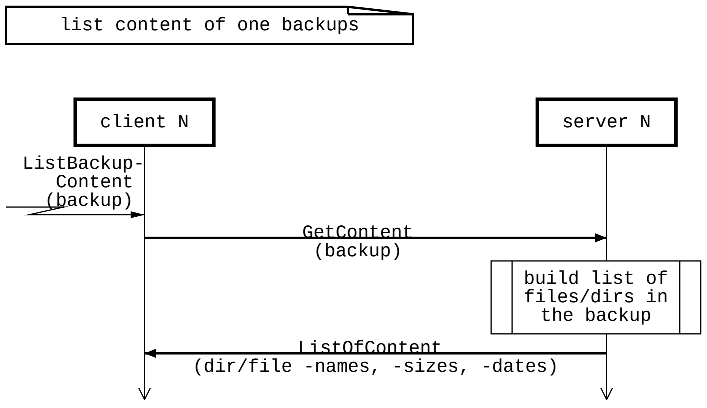

list content of one backups
client N
server N
ListBackup-Content (backup)
GetContent (backup)
build list of files/dirs in the backup
ListOfContent (dir/file -names, -sizes, -dates)

Figure 4.6: Get content of a specific backup from server

4.2 Message Flows between Client and (Graphical)User-Interface (*IFcg*)

To interact with the client process running silently in the background, the client sets up a named pipe when it starts. A Graphical User Interface (GUI) can dock to that pipe and control the client.

The messages sent through the pipe are text based commands, so the most primitive user interface will be a console with I/O redirect to the pipe.

The predefined commands are shown in Chapter 6.

4.2.1 (Re-)Connect to a server

Figure 4.7 shows how the connection between a client and the server is established from the Client GUI. If there exists a previous connection, that connection is closed before building up the new one, as every client can be connected to *one* server only at the same time.

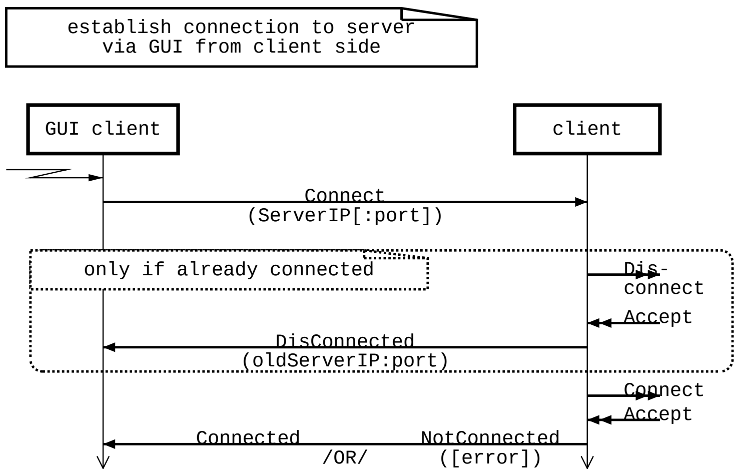

establish connection to server via GUI from client side
GUI client
client
Connect (ServerIP[:port])
only if already connected
Dis-connect
Accept
DisConnected (oldServerIP:port)
Connect
Accept
Connected
/OR/
NotConnected ([error])

Figure 4.7: Enable Connection from Client GUI

4.2.2 Start a Backup

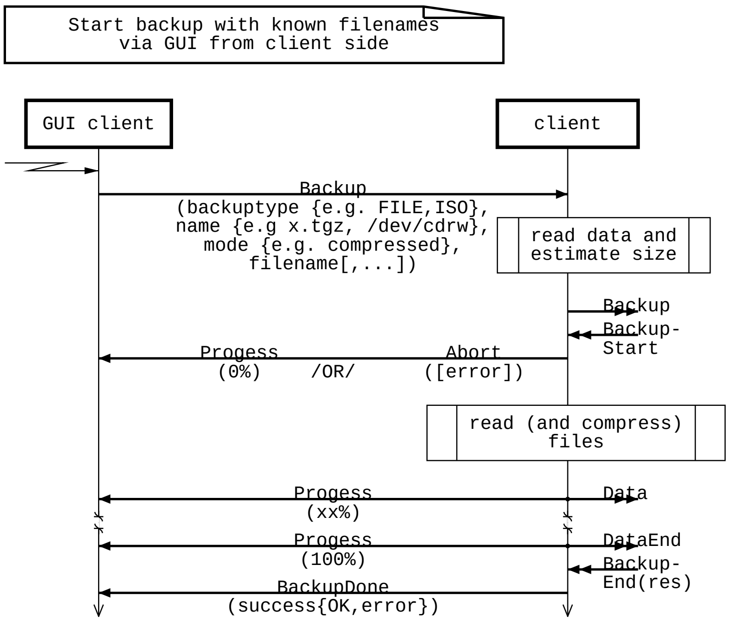
When starting a backup, the client needs to know which files to backup and some information about the backup itself: what type of backup (boot-able CD-ROM, ISO-File,...), the name/location (/dev/cdwriter, /usr/bkup/file.tgz,...) and the mode of the backup (compress_clientside=gzip,...). The message flow is in Figure 4.8.

4.2.3 Start a Restore

If a client knows (by other procedures, see chapter 4.2.4 and 4.2.5) that a certain backup exists on the server (and the client may access it), it can start restoring specified files like shown in Figure 4.9. The files can be addressed by their exact path/name (as stored in the backup) or as wildcards (path/* or path/name* – may be extended in the future).

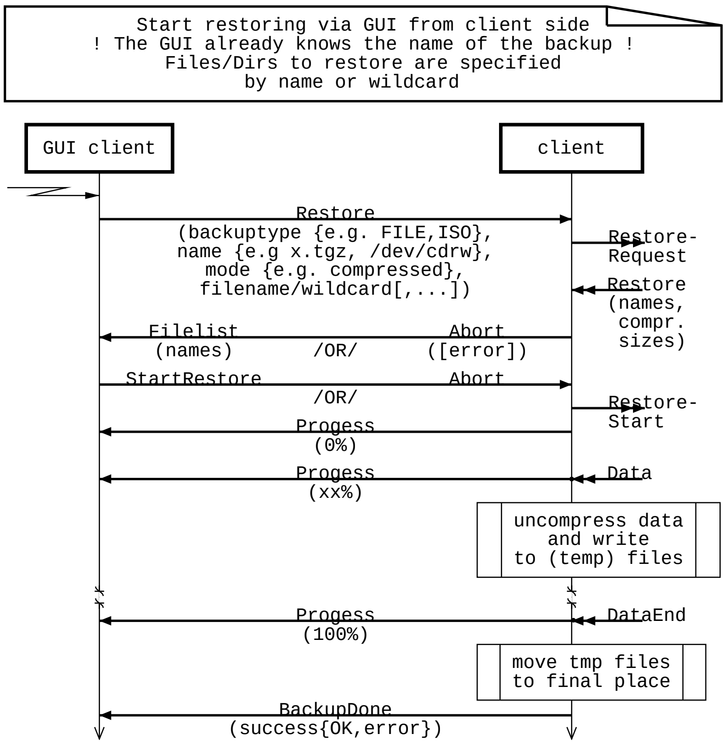
4.2.4 Get list of previous, accessible Backups

To get a list of available backups the client must ask the server which are available (maybe specific to the requesting client/user). The message flow is in Figure 4.10.



Start backup with known filenames via GUI from client side
GUI client
client
Backup (backuptype {e.g. FILE,ISO}, name {e.g x.tgz, /dev/cdrw}, mode {e.g. compressed}, filename[,...])
read data and estimate size
Backup
Backup-Start
Progess (0%) /OR/ Abort ([error])
read (and compress) files
Progess (xx%)
Data
Progess (100%)
DataEnd
Backup-End(res)
BackupDone (success{OK,error})

Figure 4.8: Start Backup from Client GUI



Start restoring via GUI from client side ! The GUI already knows the name of the backup ! Files/Dirs to restore are specified by name or wildcard
GUI client
client
Restore (backuptype {e.g. FILE,ISO}, name {e.g x.tgz, /dev/cdrw}, mode {e.g. compressed}, filename/wildcard[,...])
Restore-Request
Restore (names, compr. sizes)
Filelist (names) /OR/ Abort ([error])
StartRestore /OR/ Abort
Restore-Start
Progess (0%)
Progess (xx%)
Data
uncompress data and write to (temp) files
Progess (100%)
DataEnd
move tmp files to final place
BackupDone (success{OK,error})

Figure 4.9: Start Restore from Client GUI

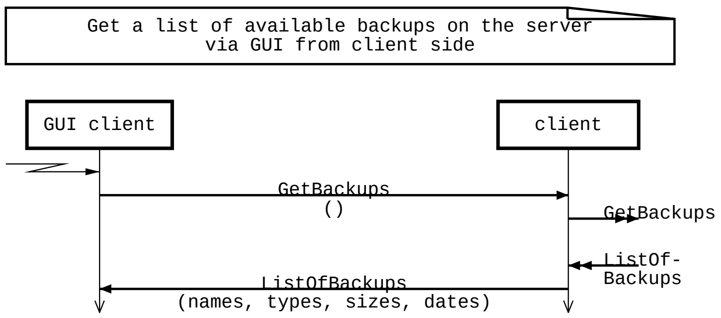


Figure 4.10: Ask Server for available backups

4.2.5 Get content (files) of a specified Backup

The client knows (maybe via the flow in Chapter 4.2.4) which backup(s) are available at the server. To view the content (the file names, -sizes,...) of a specific backup the client can ask the server to deliver this information. The message flow is shown in Figure 4.11.

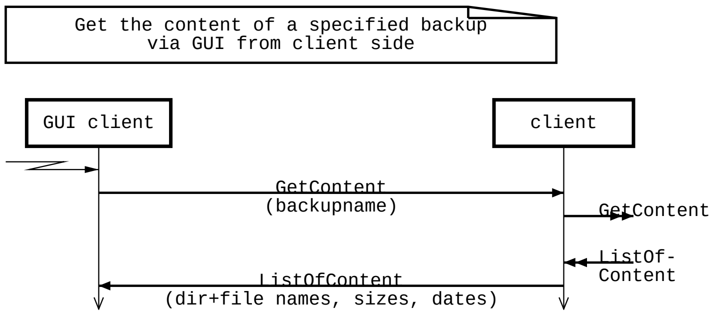


Figure 4.11: Ask Server for the content of a backup

4.2.6 Start a Verify

<to be defined>

4.2.7 Start a Compare

<to be defined>

4.2.8 Terminate Client

It doesn't make sense to close the client-server connection without terminating the client. Either the client must be connected to another server (procedure see chapter 4.2.1) if it shall do other backups/restores or the client just keeps connected to the same server (so no changes are necessary :-). Nevertheless if the client has done its job, it must be shutdown but that concerns both, the connection and the client . This procedure is shown in Figure 4.12.

Of course you can simply close the GUI of the client. But this won't have any influence to the running client...

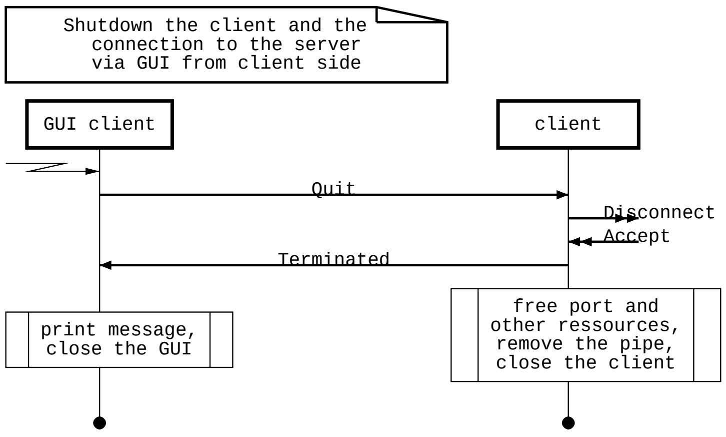

Shutdown the client and the connection to the server via GUI from client side
GUI client
client
Quit
Disconnect
Accept
Terminated
print message, close the GUI
free port and other ressources, remove the pipe, close the client

Figure 4.12: Terminate the client

4.3 Message Flows between Server and (Graphical)User-Interface (*IFsg*)

<to be defined> Ideas:

N.N. All backup/verify/compare/restore procedures that can be triggered from client side, too.

Status Show all current connections and its current status (activity).

Monitor Request server to generate (continuous) progress messages to a specified connection.

Info Server notifies the user about internal processes.

Verbose Modify the level of server's verbosity.

4.4 Message Flows between Server and its Child (*IFss* and *IFsm*)

<to be defined> Ideas:

GetClientInfo Server asks server child for info about the connection to a client (IP:port, current activity).

TerminateConnection Server asks server child to terminate the connection.

TerminateNotification Server child notifies the parent server that it will terminate now.

N.N. All backup/verify/compare/restore procedures that are requested to the server, but must be executed from a server's child.

5 Interfaces

Figure 3.1 in Section 3 (System Structure) names the defined interfaces in respect of their location. E.g. the interface between the server and the client is called *IFcs* if you see it as part of the client, and it's called *IFsc* if it is described as part of the server. This naming is continued in the software, but of course the 2 parts must fit together to transmit the information.

To describe this common part (e.g. the format of the data structures) we introduce a second naming scheme here that denominates the matter inbetween. These names are used in the sources, too. So pay attention to understand the difference: The common parts of the interfaces are named *Ixx*, their realization at the two ends are named *IFxy*.

Interface	**is connecting**
Im ("Mondo")	server/client internally with mondo
In ("Network")	client and server, *IFcs* <–> *IFsc*
Ipc ("Pipe in Client")	client to GUI, *IFcg* <–> external
Ips ("Pipe in Server")	server to GUI, *IFsg* <–> external
Is ("Server interprocess")	Server's parent instance with the child instances, *IFss* <–> *IFsm*

Table 5.1: Interfaces

5.1 Interface *Im* to Mondo

<to be defined>

5.1.1 *Im* in Server

5.1.2 *Im* in Client

5.2 Interface *In* between Client and Server

<to be defined>

5.3 Interface *Ipc*, Client's pipe to the GUI

<to be defined>

5.4 Interface *Ips*, Server's pipe to the GUI

<to be defined>

5.5 Interface *Is*, IPC of server instances

<to be defined>

6 Commands at the GUI interfaces

To interact from a Graphical User Interface (GUI) to the server or client process running locally, silently in the background, every client and the server (parent process) sets up a named pipe when it starts. A Graphical User Interface (GUI) can dock to that pipe and control the client/server.

If not otherwise specified, the server's pipe is `/var/run/monitas/server` and a client's pipe is `/var/run/monitas/client_`*`nnn`* with *`nnn`* depends upon the current connection.

(For IPv4 we could use the server's IP:port, e.g. `client_192168001001-22345` i.e. 12-digits IP address, dash, 5-digits port of the server's child for that connection).

This allows us to run more than one client on a PC (maybe more than 1 user is logged in) that can be connected to the same server (IP address) with different ports (server's child's port number) or to different servers (different IP addresses). Only one server per PC is allowed, but it can handle several connections in parallel (by several child processes). In most cases, the backup medium is a very limiting resource (only 1, 2... CD-Writer, Tape-Streamer, Database, ...) that can be managed much easier by one central instance. Otherwise we had to expand the resource management (that is already necessary between parent server and child processes) with more danger to run into dead-locks, race-conditions and other difficult-to-debug stuff.

If not denoted otherwise, the description in this chapter is valid for the messages at the client GUI *and* the messages at the server GUI.

6.1 Message Structure

The messages sent through the pipe are text based commands, so the most primitive user interface will be a console with I/O redirect to the pipe.

The used semantic is:

```
COMMAND [ARG [...]]\n
```

where

- COMMAND is a predefined (case insensitive) command, valid chars: [a-zA-Z]

- ARG is zero or more arguments for COMMAND, each COMMAND has a predefined number of mandatory arguments (some command additional might have optional arguments)
- COMMAND and ARG is separated by one space ( )
- ARGs are separated by one space ( )
- ARGs contain of at least one printable character and/or whitespace ([\t\n])
- an ARG that contains spaces must be surrounded by '...' or "..."
- inside '...' following characters must be escaped by a backslash (\) for their literal meaning:

\'	for literal single quote (')
\\	for the backslash (\) itself

  to use <c:\stefan's "quote"> as one argument use `'c:\\stefan\'s "quote"'`
- inside "..." you must not use the double quote character (")!
  There is no escape sequence defined for a literal meaning! Surprised? But this kind of definition allows us to use the ARG <c:\hugo rabson's dir\file.c> without further modification by simply surrounding it with double quotes:
  `"c:\hugo rabson's dir\file.c"`
- if the ARG itself shall begin with a double quote (") or single quote (') then quote the whole argument with '...' (and escape the ' at the beginning).
- if the ARG doesn't contain spaces but contains any quote character(s), you needn't do anything
- COMMAND line is terminated by a newline character('\n'), an optional ASCII-Null ('\0') can follow

Possible, future extension:

- Using '\0' instead of ' ' to separate ARGs (and COMMAND) will dispense with the nasty space quoting. The end of the command line is then marked by two '\0' instead of '\n'. To distinguish between old and new syntax, the client can look at the first character behind the COMMAND: if it's a '\0' the GUI uses the new syntax and all responses to the COMMAND use the new syntax, too. If the character is a space ' ' or newline '\n' we answer in old (above described) syntax and must pay attention to the quoting

6.2 Implementation Detail

There is much work that is common to all the pipe interfaces that use text based messages: extract the command, calculate the number of mandatory parameters, split the arguments, parse for escape sequences, check if an arguments fits the requested type (e.g. filename, IP address, port number,...), translate the textual argument into its machine readable format, ... And equivalent steps are necessary when we want to create and send a message. And different commands use the same argument types that always must be parsed and checked in the same way.

At the moment the extent of the complete command set cannot be given, so it would be the best to keep yet unknown extensions in mind and implement the pipe interface as a generic one:

- use tables for valid commands (different tables for server- and client-pipes [or flags in a common table to ease common syntax for equivalent server/client commands?])
- each "command" entry contains the number of mandatory and optional arguments
- each argument refers to a predefined type (not only *int*, *string* but of finer granularity like *filename*, *dirname*, *devicename*, *backupname*, *portnumber*, ...) that can be generated and checked for validity by generic routines.

The tables can easily be extended and new commands can introduced with less effort by reusing existing subroutines for the argument handling. (B.t.w. the table contains all information that is necessary for an generic help function to each defined message.) Of course this only concerns the interface handling, not the new functionality. But why reinvent the wheel twice?

6.3 Defined Commands

Table 6.1 shows the defined commands to the GUI.

Currently there are only the messages from Figure 4.7 and 4.8 entered in this table. But we recognize that the messages `notconnected`, `abort` and `backupdone` serve the same purpose. We should think about, if we want a streamline small interface (replace the 3 messages by one) or accept the redundancy. Perhaps that depends on the connected GUI. A simple command line interface (that transfers the messages transparently to the user) benefits from the different message names, a graphical user interface on the other side must join the different messages to the same "Not Done! Error" requestor.

7 Open Issues

Here I collect all the ideas that need further analysis. Some solutions may be written here if they might have side effects to the system, I must think about. Or if I hadn't have the time to insert them at the correct place ;-)

If (from/ /to GUI)	Command	No of args (mnd/opt)	type of arg 1	type of arg 2	type of arg 3	type of arg 4	type of arg 5
C/-	`connect`	1/1	ServerIP	Port			
-/C	`disconnected`	2/0	ServerIP	Port			
-/C	`connected`	0					
-/C	`notconnected`	1/1	ErrNo	ErrorMsg			
C/-	`backup`	4/*	Bkup type	name	mode	filepattern	...
-/CS	`progress`	1/0	percent				
-/CS	`abort`	1/1	ErrNo	ErrorMsg			
-/C	`backupdone`	1/1	ErrNo	ErrorMsg			
...	Table	is not	complete	...			

Table 6.1: Commands at the GUI Interfaces

- How to realize the daemon for client and server side?

  It should dock to the pipe between the GUI and the process, so it can monitor if there is something going on, and if not it can do its job.

  If a daemon triggered backup is running, no user interaction is possible, so no mix up between two backups can happen.

- Security

  What should the user be allowed? Start backups? If yes, what files? Only files, he has access to, to prevent snooping system information.

  Show contents of other backups?

  Restore files? From which backups? To which destinations?

  Backup to CD? Necessary rights to write there? Maybe handled by mondo.

  Shall the client always run as root (to access all, if triggered from outside)? If yes, how assure that no unprivileged user will abuse this?

  Shall server run as root? Server child with lower privilege? Same as user on client side - but then how to realize - user id's from client mustn't exist on server.

- Use a secure tunnel as connection? How to establish/address tunnel?

7.1 Possible Extensions

- Wildcards for specifying which files to backup/restore
  - Step 1: Standard UNIX Expressions '*' '?' for filenames not only at the end of the name
  - Step 2: also for directories (Q: does '/*/readme' match e.g. '/usr/local/share/packet/readme'?)
  - Step 3: Regular Expressions (User has circumvent above question)

- Distinguish between restore (files are written to their old places) and extract (new destination for file(s) possible).