BS Project, Computer Science, F05

Minimum Intrusion Grid Storage System

A Distributed Storage Management system
for
Minimum Intrusion Grid

Kristen Pind - <*kristen@imada.sdu.dk*> - 140671

Dept. of Mathematics & Computer Science (IMADA)
University of Southern Denmark

Date : 2005-05-31

Contents

1 Introduction

The aim of this project is to design and implement the core of a distributed storage management system, suitable for use with *Minimum Intrusion Grid (MiG)* - a Grid implementation currently under development at IMADA, SDU. The core of such a system will comprise a set of Internet hosts, preferably located on the same LAN. Each host will run a multi-threaded storage server process, that will be a member of a dynamic group - the *Storage Group*. In addition each host will run an SSL-enabled HTTP server enabling processes external to the Storage Group to communicate with the system using the HTTP protocol.
The goal is to develop a load balanced, reliable and transparent storage management system that is highly scalable and offers custom defined replication levels of individual files. That is, users of the system should be able to define the number of copies of a given file that the system must maintain.
File importing (uploading) to the storage space, will not be part of the project, mainly because this task is handled most easily by the HTTPS server process which is not part of the core of the storage management system, but also because plans exists for developing a HTTPS server application designed for MiG.
Once a given file is managed by the storage system (either because it was uploaded or because it was created by a MiG job running on some resource machine), the system will ensure that the requested number of copies of that file exists, and are kept synchronized by maintaining some essential information on the file. The storage management system is put to work when the composition of the Storage Group changes (e.g. if a server crashes or a new server joins the group) and when external processes are given access to files.
The storage management system will be implemented using the Python programming language.

2 Analysis & design

2.1 The Storage Group

2.1.1 Group type and organization

Processes running on computers can be organized in groups. Such groups can (among other things) be static or dynamic. In static groups the set of individual group members are defined before the group is initialized and cannot be changed during run-time. They are a lot simpler to implement than dynamic groups since there's no real group management. However they scale poorly and adding new hosts/processes to the group requires manual intervention. Dynamic groups on the other hand, are somewhat more complicated to implement as individual members can join and leave the group at run-time. This requires that some kind of group management must be implemented. In contrast to static groups, dynamic groups exhibit excellent scalability.
Because dynamic groups are a more elegant solution compared to static groups, but also because they are more practical in a production system where each group member may not be known in advance, the Storage Group will be dynamic.
In addition to static or dynamic, groups can be open or closed. In a closed group design, processes outside the group are prohibited in communicating with processes inside the group. Groups having the opposite property are called open groups. The fact that the group will be part of a storage management system used by MiG implies an open group: Both MiG servers and MiG resources must be able to communicate with the group.
Finally groups can be organized as *peer groups* or *hierarchical groups*. All members of the Storage Group will be equal except that one process will be chosen as the leader. I'll name this process the *Storage Group Coordinator, SGC*. This means that the Storage Group will be slightly hierarchical in nature. I'm not using a true peer group because of the way the group interfaces to the outside world, which is by means of a CGI interface managed by a HTTPS server process. MiG servers and MiG resources needs to know the *fully qualified domain name, FQDN* of a host in the storage group running this HTTPS server process. Therefore it is the responsibility of the storage server process on this host to maintain the necessary information about files managed by the system in addition to inform the other storage servers about changes to the file management information. This responsibility sets it apart from the other storage server processes in the group, and the logical consequence is to have this process handle the group management as well. That is, to let it be the SGC.

2.1.2 Group communication

Communication internal to the group can be either connection-less or connection-oriented. In connection-less communication each data packet is sent and routed individually and there's no reliability guarantied: data packets can be damaged, delayed, duplicated or lost. The Internet transport protocol *User Datagram Protocol, UDP* is an example of a connections-less communication protocol. In connection-oriented communication reliability is guarantied and every data packet is sent along the same route from the source to the destination. This route is chosen when the connection is established. An example of a connection-oriented communication protocol is the *Transmission Control Protocol, TCP*.
In order to be able to chose the most effective communication protocol for the system, it is crucial first to identify what kind of messages are exchanged, which processes communicates and how often they communicate. Also what level of reliability is required, how is crash

detection designed and what leader election algorithm is used. Conversely it could be argued, that the choice of communication protocol dictates the choice of leader election algorithm, which is discussed in the next subsection. When the group is in *idle state*, meaning that only group management communication is taking place (no file management communication), each non-SGC member periodically sends small messages, called *heart beat, HB* messages, to the SCG. These serve the dual purpose of both helping the SGC to keep track of functional group members as well as helping non-SGC processes to monitor the state of the SGC. If a connection-less communication protocol is used the SCG must send replies to each HB message. But then measures to ensure reliability would have to be implemented. If, on the other hand, a connection-oriented protocol is used, just establishing the connection is enough. No data transfer is needed (unless extra information will piggyback on HB messages). Also, at the expense of some extra overhead, using a connection-oriented protocol like TCP guaranties reliability. If the frequency of communication is very high, the overhead in TCP may result in a significant performance drop compared to UDP.
The main task of the Storage Group will be handling file information in MiG and the typical MiG-job is expected to be processor-intensive and not I/O-intensive. Therefore access to file information should be somewhat infrequent or at least not frequent enough to doubt that the performance of TCP will be inadequate. As a result the communication inside the Storage Group will be using TCP.

2.1.3 Leader election algorithm

Since one process is chosen to take on the role of leader or SGC, we need to ensure that the Storage group is never without a leader (we must also try to prevent two or more processes becoming SCG's of the group at the same time. This is discussed later). The SGC is chosen by means of a *leader election algorithm.* The goal of this algorithm is to select one (and only one) process to be SGC, and to ensure that all correct processes (processes that are functioning correctly) in the system agree on a new SGC. Two of the most widely used election algorithms are *The Bully Algorithm* (Garcia-Molina, 1982) and and an algorithm known as the *Ring Algorithm.*
In short, *The Bully Algorithm* works as follows: A process P detects that the current SGC is no longer accepting messages. P sends ELECTION messages to all processes with higher ids. If P holds the highest id or if no process with a higher id responds, P wins the election. If a higher-id process responds it takes over the election (unless it is already holding one). At some point all processes but one give up and this process is the new SGC. The new SGC then sends a LEADER message to all other processes informing them that it is the new SGC.
If the *Ring Algorithm* is used, group members are ordered in a logical ring where each member has a left and a right neighbor. When a process detects the SGC is unreachable it sends an ELECTION message to its right neighbor. The message contains a list of ids of correct processes that has seen the message so far. This list is called the *active list.* When a process receives an ELECTION message from its left neighbor and it cannot find its id in the *active list* it is appended to the list. Otherwise the process knows that it is the original sender of the message and that it has been seen by all correct processes in the group. If at some point a process is unable to send the message to its immediate right neighbor it just tries the next one to the right and so on. The original sender now uses some predetermined criteria to select one process (eg. the process with the highest or lowest id) from the *active list* to be the new SGC. Next the process sends a LEADER message (containing the id of the new SGC) to its right neighbor which in turn passes the message along to its right neighbor. After the LEADER

message has circled once in the ring the election algorithm terminates and all processes know the id of the new SGC. It is important that the criteria for selecting a new SGC from the *active list* ensures that different processes always selects the same id from identical *active lists.* Otherwise the algorithm may fail if two or more processes initiates an election at the same time.
In a stable group of small or moderate size, were members joining or leaving (and therefore elections) are rare events, the choice of a particular election algorithm is of little or no importance. If however, elections are very frequent or the group is of considerable size, there may be a small performance gain to achieve in selecting the better suited algorithm. In a group with, say 200 members *The Bully Algorithm* may prove superior, but only if we are able to utilize some form of effective broadcasting when sending messages. This implies the use of a connection-less protocol, like UDP. Of course, broadcasting can be simulated using a connection-oriented protocol like TCP, but at the cost of extra overhead. Since the Storage Group in MiG will be fairly small (initially only a handful of hosts), and because the protocol of choice will be TCP (or some protocol built upon TCP, like HTTP/HTTPS), the Storage Group will use *The Ring Algorithm* for leader elections. Once again it should be stressed, that this choice is made for esthetically reasons, not because it is thought to be much more effective.

2.1.4 Two or more concurrent group leaders

Preventing two or more concurrent group leaders is a non-trivial task. Under normal circumstances, when all the hardware functions as expected, it is relatively easy to ensure a maximum of one SGC in the group. Using a well-known leader election algorithm should work in such circumstances. The problem is when anomalies occur in process behavior or communication, which (if the system is well-designed) implies hardware problems, including network problems. One such scenario is this: Suppose that the SGC is able to send, but not receive messages. If the SGC is monitoring the other group members by expecting HB messages and using timers, it will pretty soon assume all other group members to be unreachable (which they are, from the point of view of the SGC). As it is evident that there is a risk of two leaders in this situation, the SGC should automatically give up the leader-ship.
A more difficult problem to handle is when the network topology gets fragmented. Suppose for example, that (following some router or switch failure), the network on which the Storage Group is located, is fragmented into two sections dividing the Storage Group into two subgroups. The subgroup containing the current SGC will keep that SGC, which will assume all group members now located in the other subgroup to be "dead". The other subgroup is suddenly without a leader and the remaining members will hold a leader election and thereby find a new SGC. Once again, two concurrent leaders exist. Measures to prevent such scenarios would be to take special action in the event that several group members become unreachable in a short time interval.
Since all group members (at least in the initial model) will be located on a single LAN, these problems should be extremely rare, and the design will not include special measures to deal with them.

2.2 Storage management

The storage management system provides access to files in the storage space. To prevent multiple processes from accessing the same file at the same time, it also keeps track of active file access sessions. Because MiG resources do not know the absolute path of files (but only the path relative to users home directories), a special id called a MiG session id is used in combination with the relative path name to access a file. This implies that the storage system will have to keep track of active MiG sessions. In certain situations, the storage system will also be responsible for replicating files. These situations include when server processes leave (by crash or otherwise) and when they join the Storage Group. Finally the storage system will need some mechanism to keep track of the server(s) where each file is located. To summarize, the storage management system will be responsible for the following tasks, each of which will be discussed in greater detail in the following sections:

- Providing access to files.
- Keeping track of active file sessions.
- Keeping track of active MiG sessions.
- Replicate files in case a server crashes or possibly when new servers join the group.
- Selecting (and maintaining) the list of servers associated with each file managed by the system.

2.2.1 File access sessions

When a process in the form of a MiG job is executing on some resource, it might at some point need to access a particular file. From the point of view of the running process, it is just accessing a file in the local file system, but in reality the file is located on some remote storage server. Thanks to a special library (another MiG project currently under development) that implements and (by means of a preloading mechanism) overrides some of the file access routines provided by glibc, accessing a remote file is completely transparent. The routines in this library needs to know where files can be found and as a result the storage system offers the interface functions *locate_file*, *open_session* and *close_session*. The interface function *locate_file* will most likely not be needed by the library, because the function *open_session* must be called before each file access, and because these two functions (on success) will return the same value. This value is a list of those servers where the file can be found. The list will be prioritized, which means that the second server in the list should be used only if the first is unreachable, the third should only be used if the 1st and 2nd servers are unreachable and so on.
The process of opening a file access session is shown in Figure 1. The order of each step in the process is indicated by an integer. Step `1` shows the state of the file information associated with the file `/home/MiG/user1/input.txt` when no access session is active on that particular file (as indicated by `active_session: None`). In step `2` the library code on the resource calls *open_session* passing as parameters the MiG session id (`333`), the path name relative to the home directory and the file access session type (in this case `WRITE`). To keep two or more processes from opening a file access session on the same file at the same time by calling *open_session* on different storage servers, only the current SGC will return the requested list of servers where the file can be found. The storage server that received the *open_session* call updates the file information (step `3`) and sends messages to all other members of the group telling them to update the information associated with the file `/home/MiG/user1/input.txt` (step `4`). Step `4` is required in case the current SGC crashes and another server is elected as the new SGC. It could be argued that only the SGC is required to hold information about

all the files in the storage space, but in keeping all servers up to date, we are spared of the tedious task of rebuilding the complete information, should the current SGC crash. Finally, in step `5` the prioritized list of servers is returned.

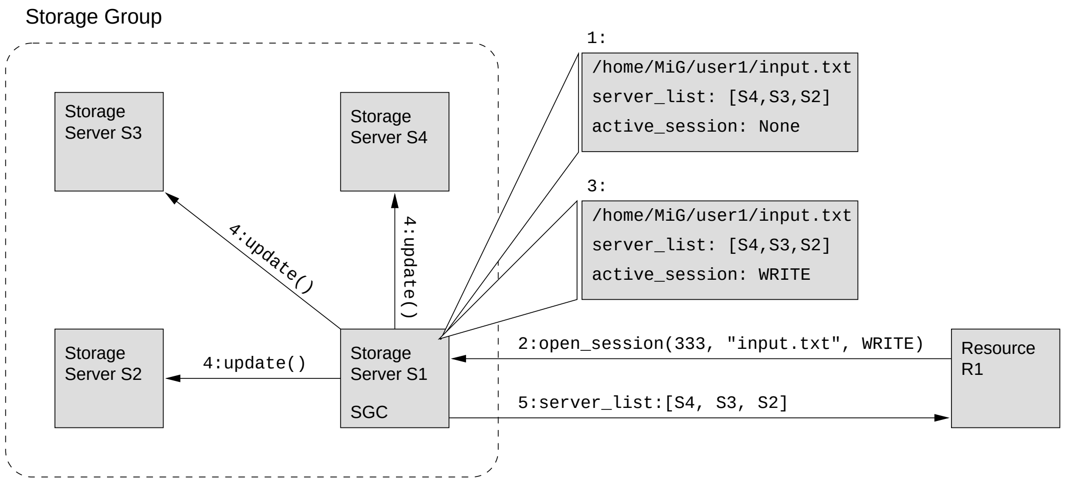
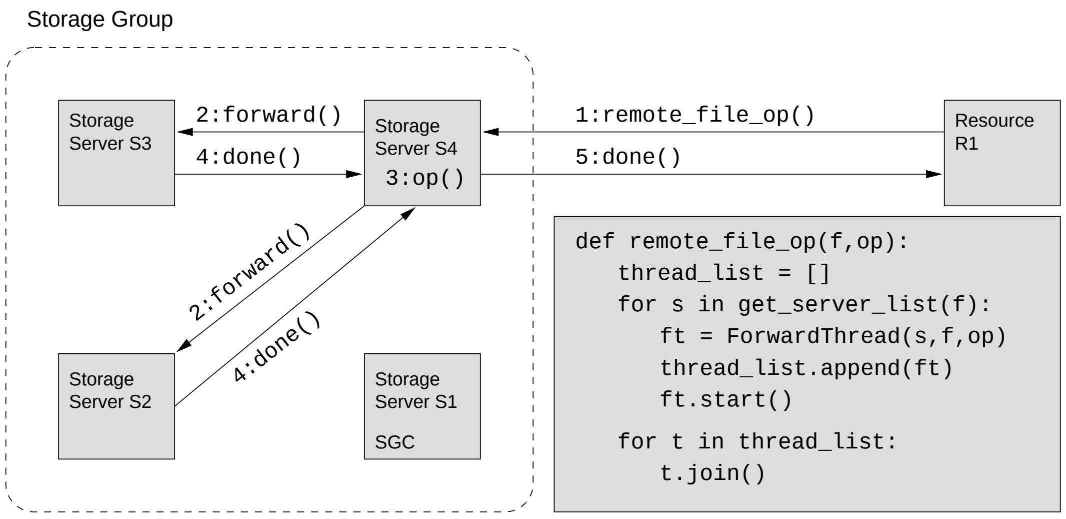


Figure 1: Opening a file access session

Once the library code receives the list of servers where the wanted file can be found it can begin performing its remote operation on the file. In the example shown in the figure, such a remote operation should be carried out on storage server `S4`. When it finishes, it is required to call the *close_session* interface function. The process of closing an active file access session is very similar to opening a session. In fact, Figure 1 would be depicting such a process if the *open_session* call were replaced by the call `close_session(333, "input.txt")`, the boxes showing the file information in steps `1` and `3` were swapped and step `5` were replaced by a return value indicating that the session has been safely closed.
In an alternative design, instead of one global state of the `active_session` field in the file information, we could opt for a local state solution. In such a design the `active_session` field would be a mapping from server ids to access session states. In fact this mapping could replace the `server_list` field, removing the need for a separate `active_session` field all together. But more important, the *close_session* interface routine would no longer be needed. Each server just closes its own session state entry in the server to session state mapping followed by a message sent to all other group members synchronizing the file information. The downside is an increase in code complexity, because now an access session is considered closed, only if it is closed for all servers in the mapping. In addition synchronizing the session state across the Storage Group (for a given file) will be a little more difficult, making the code more prone to error.
Which one is selected is probably a matter of taste, since both solutions will work. It could be argued though, that the alternative solution will have an overall better performance, because the code running on the resource is free to continue once the file operation has completed. On the other hand the typical file access in MiG is likely to be an operation on a very large file. This means, that the time spent on opening and closing a file access session will be negligible compared to the total time spent on the file operation. Because of this, and because it seems unlogical not to have to close a session once it has been opened, the storage system will use

the design shown in Figure 1 (requiring the *close_session* function call).
Even though implementing the remote file operations is not part of this project, it should be stated, that since the first design solution is chosen, the remote file operation carried out from the resource, must block until the operation has completed on all storage servers involved. This is shown in Figure 2. Because the storage servers are expected to be located on the same LAN, this requirement should not impose a performance bottleneck on file operations. If however, future storage servers will be located on different LANs and possibly geographically far apart, this may not be the best solution. In that case the second design solution should be used and a file operation should only block until it has completed on the first server.



```
def remote_file_op(f,op):
   thread_list = []
   for s in get_server_list(f):
      ft = ForwardThread(s,f,op)
      thread_list.append(ft)
      ft.start()

   for t in thread_list:
      t.join()
```

Figure 2: Executing a remote file operation

2.2.2 MiG session management

The storage system also offers the interface functions *create_session_link* and *remove_session_link*, which should be used by a MiG server process when a MiG session is created and deleted respectively. These functions creates (and deletes) a symbolic link located in the public web-space on the storage servers and points to the home directory of the user to whom the session belongs. The name of a symbolic link will be the actual session id. Such links are a necessity because MiG resources knows only the session id and the relative path to files owned by a given user. Consequently the storage system must maintain a mapping between MiG session ids and users home directories. This task is most easily performed as part of *create_session_link* and *remove_session_link*. The procedure of creating a session link is shown in Figure 3. Marking `1` shows the contents of the public web space on all servers. In step `2` the MiG server gives the order to create a symbolic link in the public web space pointing to the home directory of the user with user id `user3`. The box labeled `3` shows the contents of the public web space on the SGC after the symbolic link has been created, but before the other servers has been told to create the same link. This happens in step `4` and finally in step `5` the MiG server is notified about the outcome of the operation. The link should be created on all servers and not just on the ones serving files owned by `user3`.

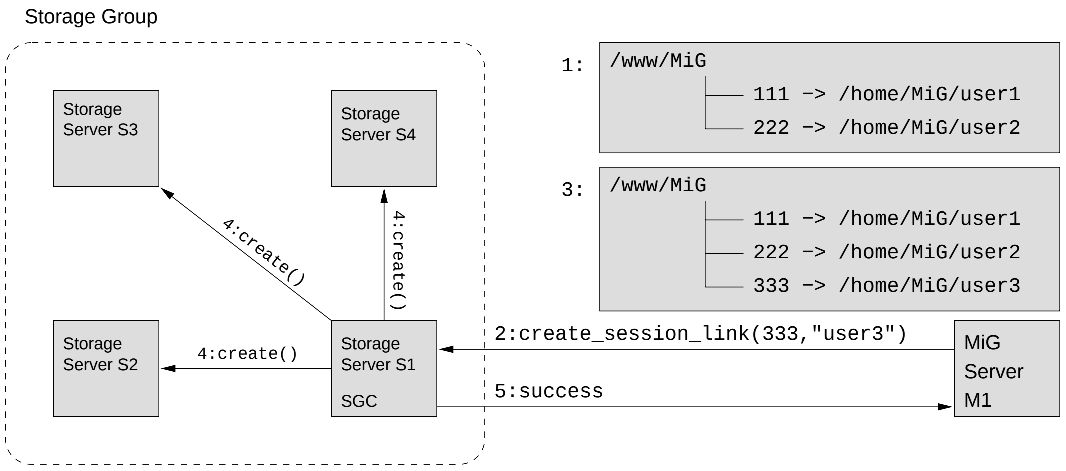


Figure 3: Creating a symbolic session link

If the link were to be created only on servers holding files owned by this user then, in the case one of these servers crashes, the system would have to recreate all the "lost" links on other servers. The users home directory, which is the destination of the session link is required to exist, because it should have been created at the point when the corresponding MiG user account were created. If the home directory is non-existent, clearly something is wrong. Parameters passed to *create_session_link* include a MiG session id and a user id. The function *remove_session_link* requires only a session id as parameter. It removes the link and mapping in the same way *create_session_link* creates them.

2.2.3 File replication

The core of the storage system is not responsible for updating every replica of a file when a file operation changes one replica. This task will be handled by a special process (probably an HTTPS process) by forwarding remote file operations as depicted in Figure 2. As this is not part of this project it will not be discussed further. However, as already mentioned, the core of the storage system will be responsible for replicating files should a server crash and possibly when a new server joins the group. In this section the term *crash* when referring to a storage server, should be understood in the broad sense that the server in question is no longer part of the Storage Group as decided by the SGC.
In the event of a server crash the SGC must inspect the information on all the file replicas that were located on the ill-fated server. For every file that were located on the crashed server, the SGC examines a special integer value called the *replication level* of the file. If this value is greater than one, it means that copies of the file can be found on other servers besides the one that crashed. The SGC then removes the id of the crashed server from the *server_list* associated with this file, and looks for a replacement server in the *member list* of the Storage Group. For a server to become the replacement server it is, of course, a requirement that such server does not already hold a copy of the file. From the subset of group members that satisfies this requirement, the one with minimum *load* is selected. This *load* metric is the key to load balancing in the storage system. The *load* metric could be a combination of several discrete "load" parameters that were updated regularly. Example "load" parameters

include disk load, CPU load, network load and locality. If in fact, a combination of two or more "load" parameters will be used, the combined *load* metric should be calculated by some kind of weight function. For example the CPU load should probably not be regarded as too important, whereas the disk load definitely should. Since it is expected that all group members (at least initially) will be located on the same LAN, and because I find the disk load to be one of most important "load" parameters (if not the most important) in a storage system, I have this parameter as the sole measure to use when balancing the system load.
A simplified example situation is shown in Figure 4. The box labeled 1 shows some information on the file `/home/MiG/user3/output.txt` before server S4 crashes. At 2 the SGC detects that S4 is no longer reachable and removes it from the group member list and from the server list of all files that were located on S4. The SGC then must select a new server to host the file. S1 (the SGC itself) and S2 are possible choices, but S2 is selected because of the load criteria (shown in the box labeled 3). Next the SGC orders S3 to copy the file to S2 (step 4), which is carried out during step 5. Finally S3 sends a message to all remaining servers with the updated information on `/home/MiG/user3/output.txt`.

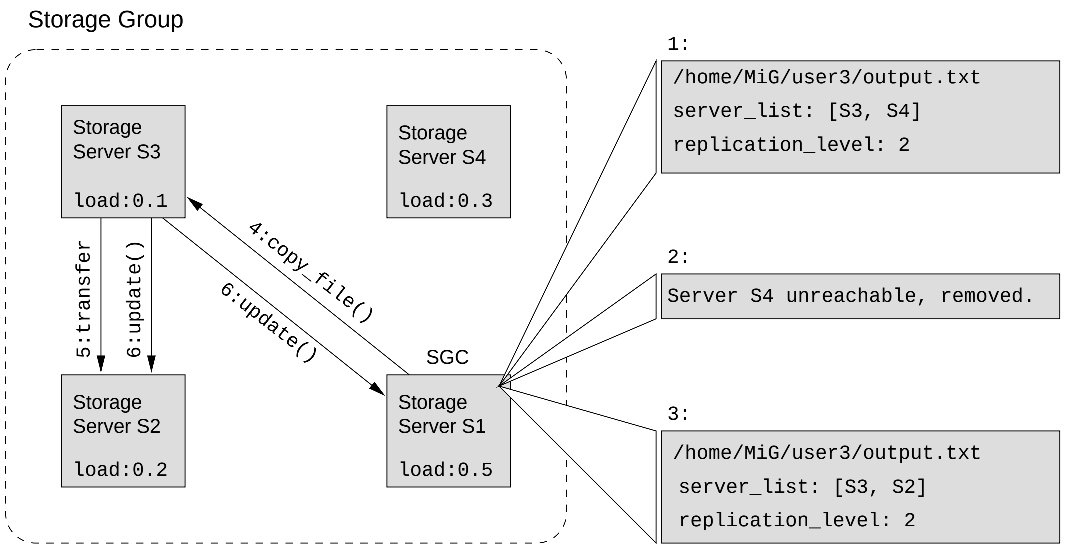

Storage Group
Storage Server S3 load:0.1
Storage Server S4 load:0.3
Storage Server S2 load:0.2
SGC
Storage Server S1 load:0.5
5:transfer
6:update()
4:copy_file()
6:update()
1:
/home/MiG/user3/output.txt
server_list: [S3, S4]
replication_level: 2
2:
Server S4 unreachable, removed.
3:
/home/MiG/user3/output.txt
server_list: [S3, S2]
replication_level: 2

Figure 4: Replicating a file in the event of a server crash

If a file with a replication level greater than the number of group members is found, the storage system has two choices: It can decrement the replication level by one or it can just leave the current replication level unchanged. If users are able to define replication levels for each file they own in the storage space (as they should be), changing the replication level for some file would also require the system to notify the user about the change. The same could be argued about not changing the replication level value, but on the other hand, it is entirely possible that the server that crashed would be restarted or even physically replaced. The important thing is to define what is being done. In the chosen model the replication level will be leaved unchanged in the event of a server crash.
A slightly different situation arises when a new server joins the group. The reason that this situation is relevant when discussing file replication, is that I chose to let the storage system leave replication levels unchanged in the event of server crashes. As a result, when a new

server joins the group, the SGC must inspect the file information, and look for files having a replication level greater than the number of group members. All such files must be copied to the newly joined server, and file information must be updated throughout the group in the usual way.
Finally note that following a leader election, the new SGC should carry out a file replication procedure similar to the one that is being carried out when a server crashes, since that is what just happened to the old SGC.

2.3 Security

An important aspect of the storage system is security and it is essential to include measures to prevent unauthorized access to files in the storage space. The most critical areas include:

- Group communication.
- The interface to communicate with external processes.
- File transfer

Inside the Storage Group the individual members are communicating with each others almost constantly exchanging information about the group itself as well as vital information about files in the storage space. File information include things like the absolute path name of files and on which servers the file can be found. Clearly, the system should try to prevent unauthorized entities to capture such information. Probably one of the best solutions is to use some kind of process identification mechanism and encrypted message transfer. E.g. to use *Secure Sockets Layer, SSL* and identification be means of signed certificates. Group communication using SSL can be achieved in numerous ways, such as by means of HTTPS or just by using TCP sockets with SSL directly. Also using signed certificates is one way of preventing unauthorized processes from joining the group.
The same solution applies equally well to communication with entities external to the Storage Group, like MiG servers and MiG resources. In fact, the interface used in such communication will be using the *Common Gateway Interface, CGI*, HTTPS and signed x509 certificates. Therefore only processes that are able to present certificates signed by MiG personnel will be allowed to access the interface operations.
File transfer refers to those situations when files will have to be copied from one machine to another (like file replication following a server crash). Of course, if the files were to be transferred unencrypted, anyone monitoring the network traffic, would be able to view the contents of these files. This must be prevented and one solution is to use the *secure copy (scp)* command, which uses SSL and is readily available. I have chosen to use the *rsync* command in combination with *secure shell (ssh)*, because in certain situations parts of the public web space will have to be synchronized on all servers, making *rsync* the command of choice. A passphrase-less private/public key pair could be used in combination with *scp* or *rsync* via *ssh* as a means of user validation and thereby removing the need for someone to enter a user name and password. If the private key is, in fact, made passphrase-less, great care must be taken to restrict access to the private key.
Another issue that should be considered is the level of local security used. It is a well known fact, that most computer related security breaches are local in origin. Since the storage servers (at least in the initial model) will be controlled entirely be the development team behind MiG, this should not be a big issue. Simply put, there will be no user accounts besides

those needed for system maintenance. Also, thanks to the MiG security model (which I will not discuss here), users will not be able to see files outside their home directory, hopefully thereby restricting the possibility of a local attack.

3 Implementation

As already mentioned, the storage system is implemented using the *Python* programming language, which is a widely used, interpreted, object-oriented language. Python supports bundling related source code files in packages and I have chosen to create such a package called *MiG Storage Element Software Package* or *msesp* to include all files that make up the core of the storage system. In addition to the *msesp* package, the implementation consists of a Python script called *storage_server.py*, which is the main storage server application. This script uses the *msesp* package and manages several threads defined therein. The storage server expects to find a configuration file called *mig_storage.cfg* in `/etc/MiG` and the server process can be started and stopped be means of a typical *System V* initialization script just called *init-script.sh* for now (can be copied to eg. `/etc/init.d/mig_storage`). Finally the implementation uses a third party SSL enabled HTTP server from *The Apache Software Foundation.* A plan exists for a native MiG HTTP server written in Python and when such a server is developed, it should be used instead. The HTTP server from Apache is configured to interpret python scripts as CGI scripts and will execute them if found at a certain location in the file system. The following subsections will present most of the implementation of the storage system in detail.

3.1 The msesp Python package

Figure 5 shows the source code files in the *msesp* package, each of which corresponds to a so called Python module. Eg. the file `message.py` located in the `net` subdirectory corresponds to a module called `msesp.net.message`. The contents of each of these files can be found attached to this report.

```
msesp
  |
  +-- __init__.py
  |
  +-- defaults.py
  |
  +-- filehandling.py
  |
  +-- functions.py
  |
  +-- globals.py
  |
  +-- group.py
  |
  +-- liface.py
  |
  +-- threads.py
  |
  +-- cgi
  |    |
  |    +-- __init__.py
  |    |
  |    +-- interface.py
  |
  +-- net
       |
       +-- __init__.py
       |
       +-- defaults.py
       |
       +-- message.py
```

Figure 5: The files making up the msesp module

The `__init__.py` files are part of a Python module setup. All there is to say about them is, that they must exist and can be used to perform module initialization steps, if need be.

`msesp.defaults`
In this module a few default values are defined, including the location of the configuration file and a Python *dictionary* that defines default configuration settings.

`msesp.filehandling`
Central to the storage system is the `msesp.filehandling` module. It defines the two important Python *classes* `FileOrganizer` and `ReplicatedFile`, which are responsible for storing information on all files in the storage space. Figures 6 and 7 shows the initialization (construct) methods of these classes.

```
class FileOrganizer:
   def __init__(self):
      self.file_dict = {}
      self.sessions = {}
      self.lock_file = forg_lock_file
```

Figure 6: The initialization method of the FileOrganizer class

FileOrganizer is essentially a wrapper class around a dictionary (`file_dict`) of *ReplicatedFile* objects (with absolute path names as keys) and a dictionary (`sessions`), that creates a mapping from MiG session ids to users home directories. As described previously, this is used when locating files given a MiG session id and a file name relative to a users home directory.

```
class ReplicatedFile:
   def __init__(self, filename, replication, grp=None):
      self.filename = filename
      self.replication = int(replication)
      self.server_list = self.assign_servers(grp)
      self.file_size = 0
      self.active_session = None
      self.session_offset = 0
      self.session_range = 0
```

Figure 7: The initialization method of the ReplicatedFile class

The `ReplicatedFile` class is a container for storing the vital file information, used by the storage system as described in detail in the *Analysis & Design* section (section 2). It should be clear what the purpose of most of the fields in this class is, but three of the fields are not used in this model though. These are `file_size`, `session_offset` and `session_range`. I have chosen to define them because they could be used in future extensions to the current implementation. Eg. the `file_size` could come in handy in the case of file migration triggered by low disk space, since in such an event, the best effect is achieved by migrating large files. The other two fields could be used if shared access to files were to be allowed. In addition to the init methods the classes defined in `msesp.filehandling` offers various auxiliary methods.

`msesp.functions`
This module defines quite a lot of auxiliary functions used throughout the *msesp* package and by the `storage_server.py` script itself. These include a function to load configuration

definitions, a function to handle logging and many more.

`msesp.globals`
Defines a few variables that are considered global to the complete *msesp* package and logically fits nowhere else.

`msesp.group`
Central to the "group" part of the implementation, this module defines the `ServerGroup` class consisting of several methods and fields (Figure 8). It is the primary data structure used by some of the threads defined in the `msesp.threads` module as well as the main storage server script, to manage the dynamic group of storage servers.

```
class ServerGroup:
    def __init__(self, my_ip=None):
        self.members      = [my_ip]
        self.leader_id    = my_ip
        self.my_id        = my_ip
        self.election     = FALSE
        self.election_started_at = 0
        self.timers       = {}
        self.loads        = {}
        self.members_lock = Lock()
        self.timers_lock  = Lock()
        self.elect_lock   = Lock()
```

Figure 8: The initialization method of the ServerGroup class

Interesting fields defined in the *ServerGroup* class, include `members` - the list of members of the storage group, `timers` - a mapping from member ids to ids of associated *heart beat timers* and `loads`, which is a mapping from member ids to disk loads. But also the fields `election` and `election_started_at` deserves to be mentioned. The field `election` is used to indicate weather an election is currently in progress. This is used by several threads enabling them to take appropriate actions once they detect that an election has been started. The other field - `election_started_at` - are used in some special situations to determine when the last election was started. For example the *HeartBeatSender* thread uses this to determine if an election was started during its sleep.
Each group member is identified by its IP address, so `members` is a list of IP addresses and the keys of both `timers` and `loads` are IP addresses. The values of the `timers` dictionary are *TimerThread* objects defined in `msesp.threads`, so each group member is associated with one *TimerThread*. This is used by the SGC to monitor the state ("dead" or "alive") of group members.
Finally the values of the `loads` dictionary are disk loads as reported by each group member itself. The load is a floating point value describing the fraction of used disk space on the partition that holds the MiG users home directories. So if the load is 0.4, say, it means that 40 percent of the available disk space is in use. This is the sole load balancing measure used, and is implemented using the Python `os.statvfs` and `statvfs` modules. The function *get_disk_load* calculates the disk load (or more correctly the partition load) and is located in the `msesp.functions` module. Each group member reports its disk load as a value piggybacking the HB messages it sends to the SGC.

Methods of the *ServerGroup* class include methods to insert and remove members and methods to manage timers among others.

`msesp.liface` and `msesp.cgi.interface`
The functions defined in `msesp.liface` are *low level* versions of the ones offered via the CGI interface as handled by the `msesp.cgi.interface` module. In truth `msep.gci.interface` is not a Python module, but rather just a Python CGI script. The functions in `msesp.liface` are therefore called by the code in the CGI script. As an example, suppose that some MiG server wants to create a symbolic session link, it uses a URL having the following format to access the correct interface function:

```
https://<IP of storage server>/interface.py?fname=create_session_link&\
              sessionid=<session id>&user=<user name>
```

The code in `msesp.cgi.interface` detects what interface function has been requested, performs a few parameter checks, calls the corresponding low level version of the function located in `msesp.liface`, and prints the return value to stdout (and thereby returning the result to the client process). The reason for implementing the low level version of the functions, and not just let the CGI script handle it all, is primarily that some part of the storage system might itself need to call one of these functions. Having low level versions - actually true Python functions - enables threads that are part of the storage system to call these functions in an easy way. Also it makes the code more modular and more readable.
The following pseudo prototypes corresponds to the functions that is offered via the CGI interface:

- `list locate_file(session_id, filename, config_parser)`
- `list open_session(session_id, filename, session_type)`
- `boolean close_session(session_id, filename)`
- `boolean create_session_link(session_id, user)`
- `boolean remove_session_link(session_id)`

`msesp.threads`
This module defines several thread classes, that are used by the core storage system, to perform various tasks. These classes includes a *Listener* thread accepting incoming TCP connections on a specified port, a *ConnectionHandler* thread to handle the incoming requests, a *HeartBeatSender* thread that periodically informs the SGC that the server process is alive (implying that this thread is started, only on non-SGC group members) and special *timer* threads used by the SGC to monitor the health of each group member. Also defined in this module are two *Replicator* threads - one to handle file replication in the event of a server crash (*PostCrashReplicator*) and one to handle replication when a server joins the group (*PostJoinReplicator*).
Since the implementation utilizes a third party HTTP server (apache), some kind of *Interprocess Communication* (IPC) mechanism is needed in order to enable the HTTP server and the storage server processes to communicate. Such communication can be implemented in various ways, such as via sockets or pipes. I have chosen to use fifo files, and as a result `msesp.threads`

also defines a *FifoServer* thread. Interface functions defined in `msesp.liface` then uses the auxiliary function *retrieve_info*, defined in `msesp.functions` to communicate with the *FifoServer* thread.

`msesp.net.defaults` and `msesp.net.message`
The first of these defines a few values, such as time interval between each HB message sent and *timeout* values. The second module defines the *Message* class, which is used in group communication. The initialization method of the *Message* class is shown in Figure 9.

```
class Message:
    def __init__(self,type,data=None):
        self.type = type
        self.data = data
        self.chunk_size = 4096
```

Figure 9: The initialization method of the Message class

The `type` field is used to indicate if a given message is a request to join the group, a HB message etc. Possible message types are also defined in the `msesp.net.message` module and listed in Figure 10. From previous discussions the role of most of these message types should be obvious. The `data` field is used for holding the actual message data and the `chunk_size` field is used by the *send* method of the class. This method actually both sends the message to some recipient and tries to receive a reply. So maybe a more appropriate name could have been chosen. When the *send* method is receiving, it tries to receive `chunk_size` bytes at a time.

```
JOIN             = 80
LEAVE            = 81
ELECTION         = 82
LEADER           = 83
HEART_BEAT       = 84
ARE_YOU_ALIVE    = 85
WELCOME          = 86
GOODBYE          = 87
REJECT           = 88
REDIRECT         = 89
GROUP_UPDATE     = 90
COPY_FILE        = 91
FILE_UPDATE      = 92
SESSION_UPDATE   = 92
CREATE_SYMLINK   = 93
REMOVE_SYMLINK   = 94
```

Figure 10: Possible message types as defined in `msesp.net.message`

As can be seen from the source code, group communication in the Storage Group uses plain TCP sockets, which is insecure and not suitable for a production grade storage system. One solution is to extend the CGI interface to include group communication and then use *pycurl*, which is a Python wrapper around *libcurl*. This seems slightly clumsy though, because then IPC would have to be used between the interface script (running inside apache) and the storage server process. I find, that a more elegant solution would be to keep the plain TCP sockets version, but with SSL added. It seems that there is a few Python modules that can accomplish this. In my opinion, the most promising is called *TLSlite* and how simple (at least in theory) it is to use is demonstrated in Figure 11.

```
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('', 4444))
sock.settimeout(10)
connection = TLSConnection(sock)
connection.handshakeClientCert()
connection.write(Message(HEART_BEAT, None))
connection.close()
sock.close()
```

Figure 11: A template for TCP socket communication using TLSlite

3.2 The main storage server process

As previously stated, the main storage server process (implemented by the Python script *storage_server.py*) can be stopped and started by means of a *BASH* shell script, called *init-script.sh* at the time of testing. This script could be copied to, say, `/etc/init.d/mig_storage`, and the main storage server process could then be started, respectively stopped using the following commands:

```
/etc/init.d/mig_storage start
/etc/init.d/mig_storage stop
```

The main storage process is basically made up of an initialization procedure followed by an endless loop. During the initialization the process will try to get the IP address of the current SGC by contacting the main MiG storage HTTP server, which is hopefully running. If the IP address of the current SGC was successfully retrieved a *JOIN* message is sent to this IP address, and if it is indeed the SGC, a *WELCOME* message will be returned. Piggybacking this message is a *FileOrganizer* object and a special container object from which a *StorageGroup* object is built. The initialization procedure then checks the `sessions` dictionary in the *FileOrganizer* object creating home directories and symbolic links as needed. Finally the procedure tries to save the *StorageGroup* and *FileOrganizer* object instances to disk.
If, on the other hand, the initialization procedure was unable to get the IP address of the current SGC, what happens next should be determined by the configuration directive *leader_if_sgc_unreachable* in the following way: If the value of this directive is 'yes', the procedure should create a new storage group and become the initial SGC. If the value is 'no', the process should terminate. This has not yet been implemented, but by the time of deployment it should be.
Note that in this initial model, the FQDN of the main MiG storage HTTP server is statically fixed to one particular machine. This is, of course, a single point of failure, but this problem should be eliminated when a native Python MiG HTTP server is developed, because a such should be implemented as a distributed HTTP server.

3.3 Storage server configuration

The storage system supports a limited custom configuration by means of an external configuration file that should be located at `/etc/MiG/mig_storage.cfg`. The storage server implementation uses the Python ConfigParser module to parse and load the configuration from this file. Using this configuration file, custom file locations (like the locations of log and pid files) can be defined and also things like the default replication level to use when new files are created in the storage space and what log level to use.

3.4 HTTP server process

The Apache HTTP server can be SSL enabled by using an Apache module called *mod_ssl* or simply by using a special version of Apache HTTP server called Apache-SSL. It makes no difference which one is used, so I chose Apache-SSL, because it is available as a separate package for the Linux distribution I happen to use at the moment of testing.
The directory `/var/www` is configured to be the public web space of the HTTP server, and the directory `/var/www/MiG` will be the location of the MiG session id symbolic links.
The Apache-SSL HTTP server had to be configured to allow executing Python scripts. This was accomplished by using a *ScriptAlias* directive in the Apache configuration file (httpd.conf). Also the environment variable *PYTHONPATH* had to be defined inside the Apache configuration to point to the location of the *msesp* package.
Signed certificates can be used with Apache-SSL to verify the identification of clients, but I did not use this functionality. It (or something similar) should, however be used when the system is deployed.

4 Tests

Because this project only deals with the core of the MiG Storage Management system, with key parts (like file uploading) missing, no full scale test has been performed. Instead tests have been done manually and by simulating certain situations. The tests can be divided into two parts: The ones that are related to the operation of the dynamic group and the ones that deal with Storage management. Sample output from some of the tests are included as described with each test. The output are in the format of raw log files generated by the storage server process. In the tests, three ordinary workstations, running a Debian GNU/Linux based operation system, were used. They were configured with the following IP addresses:

- 192.168.194.44
- 192.168.194.77
- 192.168.194.99

Since the criteria for being elected SGC is to have the lowest id (IP address), it should be noted that the above addresses are listed with the lower ids being at the top of the list.

4.1 Group-related tests

Most group-related tests involve a leader election in some way. The classic example is if the current SGC crashes, but also when new servers join the group a leader election is carried out.

4.1.1 Leader election

In order to test the implementation of the *Ring Algorithm*, a storage server process were started on each workstation in turn while monitoring the output from the logging mechanism. As expected, every *join* event resulted in a leader election and each time the same storage server process were elected SGC. Printed output from this test can be found attached to the report. Each log file is named according to the format `<IP>-join.log` where `<IP>` is the last two digits of the IP address of the machine on which the server process were executing.
An SGC crash were simulated by manually stopping the current running SGC and monitor the log file. A leader election were started by one of the two remaining group members and the one with the "lowest" IP address were elected the new SGC. Output from this test can be found attached having file names of the format `<IP>-sgc-crash.log` (since 44 was original SGC, the output is from 77 and 99).

4.2 Storage-related tests

Storage-related tests include testing all interface functions, testing file replication following a server crash and file replication following the joining of a new server.

4.2.1 Testing interface functions

The interface functions were tested manually by using a HTTP client to "visit" URLs corresponding to calling each function. In fact, I created an HTML document having links to interface functions in a top frame of the browser window, with the results returned presented in a bottom frame. All functions worked as expected.

4.2.2 File replication following a server crash

To test how the system handled such an event, the replication level of some test file were set to 2 and the file were manually copied to the two servers chosen by the storage system (and listed in the `server_list` of the corresponding *ReplicatedFile* object instance). One of these servers were then stopped to simulate a server crash and the log output were monitored. The system, naturally selected the last remaining server as a replacement server. The file were correctly copied to that server and file information were updated as expected on the remaining servers. Of course, it would have been a more robust test if there had been more than just one server to chose from, but since the selection algorithm simply sorts the possible replacement servers by disk load and selects the first one (the one with minimum disk load), it would without a doubt also work as expected with more servers to chose from.

4.2.3 File replication following a join

In this test a test file having a replication level of 3 were created with only one server running. Then letting first the second, followed by the third server join the group, it could be observed that the file were copied to both of these servers in turn and that the file information were updated on all group members. Exactly as expected.

5 Conclusion

Because of the lack of full scale tests, this implementation should, of course, not be considered for a production system before such tests can performed. The tests that were carried out did show, however, that the implemented model is feasible. The group management, including using the implementation of *The Ring Algorithm* for leader election, works quite well as demonstrated by the tests. Dynamic group events such as when servers join and leave the group unfold as expected and servers becoming unreachable were detected in every single test. That is, every simulated SGC crash were correctly detected by one (or possibly more) of the group members and every simulated crash of a group member were correctly detected by the SGC. The tests also show that preventing more than one SGC at the same time, works in ordinary circumstances, which should be enough in the initial model.
The tests also indicate that the design and implementation of the storage management system works as expected. The choice of offering replication of individual files results in a simple, yet powerful system that seems to work well. As demonstrated, file replication in the event of a server crash or a server join, works without problems as do all the functions available to external processes via the CGI interface.
There could, however, be a problem with the IPC mechanism used by the CGI interface (running as part of apache) to get information from the storage server process. As mentioned the chosen IPC method uses fifo files and a few times during testing, the process that initialized an IPC session did never receive any reply from the *FifoServer* thread, therefore blocking forever. This should not be any problem, because when deployed the storage system will, hopefully, be using an HTTP server running as a thread inside the storage server process, thereby eliminating the need for IPC.
The tight security model, should help prevent unauthorized access to information and files inside the storage system. Elements of this model include both client and server identification using signed certificates and even though I have only tested the system using a server side certificate, it is a technique that is known to work. Another matter is the security of the group communication, which at the moment uses plain TCP sockets and no identification mechanism. But, as discussed in the *Implementation* section, possible solutions to this problem has been explored, but not yet fully tested.