Ruprecht-Karls Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Internship

Optimisation for Small Files in PVFS2 - Integrating File Meta Data into Directory Entries

Name:	Kai-Hajo Husmann and Andreas Beyer
Supervisors:	M.Sc. Julian Kunkel, Prof. Dr. Thomas Ludwig
Date of submission:	December 12, 2009

Abstract

This internship originated in the problem of optimising PFVS2[1] for small files through the integration of file meta data into directory entries.

A PVFS independend solution for that problem, the so called container format, had been developed by Hendrik Heinrich[2] already. Out of this the idea was born to implement a container feature (CF) into PVFS[3]. Since it was not possible to integrade the H. Heinrich code into PVFS we decided to concentrate on its main issue: That is the sparing of redundant meta data.

Unfortunately the speed evaluation of our code rather discourages from an implementation of a container feature for PVFS - at least if there is not planned a much higher degree afford for solving that problem. But with our design some states (within the client side state machines) can already be spared and with futher coding a reliable speed enhancement for containers will surely be possible. The meta data of the files within a container can be spared completely. It will be there only one time as meta data for the container itself. Therefore the smaller and numerous the files are, the bigger the enhancement considering hard disk usage. For a few big files this code will never be of much use - but it doesn't aim at that.

Authors

This internship was a team working project by Kai-Hajo Husmann and Andreas Beyer. Whilst the coding of the container implementation was done in team we later separated our working fields with A. Beyer programming the MPI Test Code and preparing the presentation and K. Husmann [me] writing this documentation.
If you want to contact me feel free to mail me: `mailto:kai.husmann@googlemail.com`

Contents of this Document

This document contains all steps that where done in our internship.

1. setting up of a development plattform for working with PVFS
   (Linux, Git, Eclipse, PVFS, MPI)
2. a possible design or rather some notes about that
3. imlementation of a prototype
4. evaluation of the resulting PVFS-CF code
5. diverse notes about future work

The content does not include the actual coding of a full functional container implementation. That still has to be done. But for that purpose this document will surely be of some use.

[1]PVFS is the abbreviation of Parallel Virtual File System and it is a code developed by `http://www.pvfs.org` under General Public Licence

[2]Hendrik Heinrich is a member of our working group and his bachelor thesis was a container format, which mainly packs the data of similar files into one file thereby sparing redundant meta data and internal fragmenting. Its advantage to standard archivers is that it fully integrates into the EXT3 file system

[3]Therefore, our code derived out of this internship is called PVFS2-CF

Contents

1 Introduction

1.1 Motivation and Basics

At the time we begun our internship the “Heidelberger Krebsforschungszentrum” (Cancer Research) had been into various researches were they took lots of of high resolution microscope pictures. Later on some of these pictures are planned to be refactored into a movie. During this research millions of pictures had already been taken. The total data is in order of terabytes.

Handling such a big amount of files brings along some problems, one of them bad scaling of the simple listing of these files. This is because standard file systems have to meet many different needs - which normally do not include the handling of many (similar) files -, and therefore cannot be optimised for that kind of job. To cope with that a special container format was developed in Heidelberg by Hendrik Heinrich. It extracts the coherent meta-data of each file and bundles it within the containers meta-data. The single files are packaged one after the other in one big file similar to an archiver. A difference to a standard archiver is that it fully integrates into the Linux file system and supports standard file system operations like ls, e.g..

One knows that the hard disk speed is the bottleneck for I/O intense workloads. Therefore the size of high resolution pictures and the time needed to load lots of them into RAM poses another problem. Here a parallel file system brings some relief.

But fortunately there exists another solution: Parallel file systems. One representant is the *Parallel Virtual File System*, available in version 2, abbr. PVFS. Files which are stored in PVFS will be segmented and their segments will be distributed among any number of so called *data servers*. This process will again be managed by the so called *meta server(s)*. When a client wants to load a file it asks any of the meta servers how its segments were distributed and finally retrieves some segments from each data server and puts them back together in local RAM. Therefore the network connection becomes the new bottle neck of loading speed.

So this is why this internship came up: *Can we improve meta data performance by embedding the container format into PVFS?*

1.2 Assignment

Since a release kind of implementation would have been far to much workload for an internship we had to trim our plans and before such a big project can be tackled one has to evaluate its use. Therefore we decided to implement just enough functionality to evaluate the use of a PVFS container. That’s why our code is rather sketchy - it shall serve only the purpose of evaluation. For that evaluation one has to compare how certain operations scale with the original PVFS against an estimation of a PVFS container, which our sketchy code proposes. Whilst the process of implementing it became clearer each day that our original plans had to be ajusted. Finally we downgraded our demands and decided it would suffice to implement the container feature just within these two intern PVFS-calls:

- create - create empty files within a container
- lookup - find files within a container, `ls` for example

With the bundling of the meta data and the stinting of inodes how much faster can the creation and listing of files get?

1.3 First Steps

Before one begins with such an assignment it is useful to decide on the tools and materials to use. The first thing to choose is the code basis on which to operate. Well, not to rewrite the PVFS code was quite obvious!

We then decided not to use the above mentioned container code because the PVFS code does not have a good interface to implement other code “plugin-like”. And again the structures of each code were very distinct. So we started with the recent release of PVFS - at that time being PVFS 2.7.0. -, written in C and using a state-machine precompiler.

This was our road to take:

- understand the state machine precompiler
- remember the advantages and obstacles using *c* programming language
- sniff out the PVFS code and uncover its secrets
- understand the PVFS-create state machine and others
- find the right positions for our changes
- implement - hack - test - and correct the written until it works

But before one can begin with the above road trip the "cars" have to be chosen, the environment to be put up and so on. The next chapter will bring up the other tools needed for such a development which are in short a development platform, a version control system, a testing environment and something for the documentation.

1.4 Further Reading/ Documentation

1. PVFS Documentation
   `http://www.pvfs.org`
2. MPI Documentation
   `http://www.mcs.anl.gov/research/projects/mpich2/`
3. From our working group: Parallele und Verteilte Systeme

   Julian Kunkel Master Thesis

   Hendrik Heinrich Bachelor Thesis: Container format
   `http://pvs.informatik.uni-heidelberg.de/theses.html`

2 Software Environment

This section describes the preparations needed to configure a useful and effective software environment that can cope with the requirements of a project like ours. It does not contain any super secrets; just some tips, tricks and howtos. But it is my considered opinion that it can still be quite useful when preparing to work with the PVFS code.
The following lists a short overview of all needed *tools* and thereafter a subsection for each part will describe that in particular:

1 Version Control System Git: http://git.or.cz/

2 Original PFVS Source PVFS 2.7.0.: http://www.pvfs.org/

3 Development Platform Eclipse: http://www.eclipse.org/

4 Parallel Execution MPI: http://www.mcs.anl.gov/research/projects/mpich2/

5 Test Environment During this internship tests were only performed on a local machine. Tests on a cluster are future work (see section 5.1.10 at page 26) and therefore are not described any further in this section.

2.1 Version Control System: *Git*

In any informatics project a version control system is a useful tool. But the fact that we're a two members team in this project makes it a must-have. Without such a tool the merging of changes by different programmers - especially within a single file - poses quite a problem. And then there is the advantage of undoing false commits and reviewing changes.

Due to the fact that CVS is somewhat out of date we had to choose between Subversion (SVN) and Git. The difference between Git[4] and SVN[5] is mainly that Git keeps a local copy of the common repository and therefore differentiates between commit and push. An advantage of SVN would have been the better integration into the Eclipse Plattform[6] due to the quite handy and effective plugin called Subclipse[7]. But we chose Git because a git-server was up and running already.

The most common Git commands are listed below:

1. To create a local copy of the server-side-repository:
   `git clone ssh://user@server:serverside-location`

2. Before any working day begins one has to update his own repository and branch:
   `git pull` will do just that. It will merge the server side repository changes into the local repository status. Most changes can be merged automatically but if both repositories have changes in the same line (e.g.) it will bring up a merge-conflict. This then has to be fixed first, followed by a git commit and push (described below).

3. Changes have to be committed to the local repository:
   `git commit -a` submits all changes to supervised files, new files first have to be added using
   `git add filename`.
   If the `-a` flag is ommitted only files that have been explicitly prepared to be committed by
   `git add filename` are committed.

   NB: If `git commit` is used probably more information about what will be committed is wanted. `git status` will list exactly that information: files to be committed, changed files, untracked files. `git commit -a` will show all files that the equivalent `git commit -a` would update.

[4]available at `http://git-scm.com` under GPL
[5]available at `http://subversion.tigris.org` under GLP
[6]available at `http://www.eclipse.org`, under EPL by Eclipse Foundation, Inc.
[7]from the same group as SVN

4. At least the command `git push` will push all commits onto the server-side repository. In case a colleague has pushed his code recently it will ask you to perform a `git pull` to merge his changes into your local ones before you can commit and push your (now merged) code again. So you can test the merged code before you commit it.

Therefore if Git is used thoughtfully it will hinder any destructive changes to the server-side repository. And it is just a great help to multi-user-development.

2.2 PVFS

At start up Michael Kuhn prepared our repository containing the source of PVFS 2.7.0. But the source alone is not sufficient. We also had to install some packages before installing PVFS. Using aptitude in an Ubuntu environment this was done by calling the following command:

```
~$ sudo aptitude install build-essential libdb4.5-dev bison flex
```

The second thing was to prepare a useful directory structure and to clone the server-side repository:[8]

```
1 ~$ mkdir pvfs2
2 ~$ cd pvfs2
3 ~/pvfs2$ mkdir build
4 ~/pvfs2$ mkdir inst
5 ~/pvfs2$ mkdir src
6 ~/pvfs2$ cd src
7 ~/pvfs2/src$ git clone ssh://user@repo-url
```

The folder `build` is to contain all files needed and created during build meanwhile the folder `inst` is to contain the installed *PVFS program*. In `src` we kept our repository which does not only contain source files but also our documentation, configuration scripts and so on. Therefore it would have been better to to replace lines 5-7 by something like

```
~/pvfs2$ git clone ssh://user@repo-url ./myrep
```

This would have created the folder `myrep` and filled it with the repositories content. And would have served us as a better directory structure. Since then we would have had
`~/pfvs2/myrep/src` instead of `~/pfvs2/src/praktikum-pvfs/src`
which actually looked a bit annoying. Anyway..

Next step was to install and to configure PVFS - in our case:[9]

```
~/pvfs2/src$ cd ../build
~/pvfs2/build$ ../src/praktikum-pvfs/configure --prefix=/home/husmann/pvfs2/inst
~/pvfs2/build$ make
~/pvfs2/build$ make install
```

followed by the creation of a file called `pvfs2tab`, filled with the following contents:

```
tcp://localhost:3333/pvfs2-fs /pvfs2 pvfs2 defaults,noauto 0 0
```

This does describe the location of the PVFS server at first (`tcp://localhost:3333/pvfs2-fs`), then its mounting point (`/pvfs2`) and finally some flags (`pvfs2 defaults,noauto 0 0`) probably describing the server and connection in some way - just copy those!
This file is for later use keep it in the `~/pvfs2/` folder.

[8] user and repo-url have to be replaced:
user: beyer or husmann or rather your account
repo-url: pvs-cluster.informatik.uni-heidelberg.de/home/sighpio/Git/praktikum-pvfs

[9] /home/husmann has of course to be changed to /home/your-linux-account

Now we can begin with the server configuration. Therefore we have to create just another file: `pfvs2-fs.conf`. There are two ways to create this file, either by hand or using the `pvfs2-genconfig` tool which can be found in `pvfs2/inst/bin`.[10]

In our case it was easiest to use a hand-created file with the following contents:

```
<Defaults>
    UnexpectedRequests 50
    EventLogging server
    LogStamp datetime
    BMIModules bmi_tcp
    FlowModules flowproto_multiqueue
    PerfUpdateInterval 90000
    ServerJobBMITimeoutSecs 30
    ServerJobFlowTimeoutSecs 30
    ClientJobBMITimeoutSecs 300
    ClientJobFlowTimeoutSecs 300
    ClientRetryLimit 5
    ClientRetryDelayMilliSecs 2000

    StorageSpace /dev/shm/husmann/
    LogFile /home/husmann/pvfs2/pvfs2-server.log
</Defaults>

<Aliases>
    Alias localhost tcp://localhost:3333
</Aliases>

<Filesystem>
    Name pvfs2-fs
    ID 2134887975
    RootHandle 1048576
    <MetaHandleRanges>
        Range localhost 3-4611686018427387904
    </MetaHandleRanges>
    <DataHandleRanges>
        Range localhost 4611686018427387905-9223372036854775806
    </DataHandleRanges>
    <StorageHints>
        TroveSyncMeta yes
        TroveSyncData no
    </StorageHints>
</Filesystem>
```

[10]You can read more about this in REPOSITORY/doc/pvfs2-quickstart.tex or on the website documentation: `http://www.pvfs.org/documentation/` –> Install Guide

Important parts

- default:

```
StorageSpace /dev/shm/husmann/
LogFile /home/husmann/pvfs2/pvfs2-server.log
```

- Aliases

```
Alias localhost tcp://localhost:3333
```

- Filesystem

```
Name pvfs2-fs
```

The above mentioned important parts have to be fit to your needs. Probably your account isn't husmann for example. And the other things should be self-describing just as well.

Now I advise you to create two more files to start and stop the PVFS server. The files were created in the `~/pvfs2/` folder and do contain the following:

- start.sh

```
#!/bin/bash
./stop.sh
rm -rf /home/husmann/pvfs2/inst
make -C build -s install
./inst/sbin/pvfs2-server ~/pvfs2/pvfs2-fs.conf -a localhost -f
./inst/sbin/pvfs2-server ~/pvfs2/pvfs2-fs.conf -a localhost
./inst/bin/pvfs2-cp pvfs2tab /pvfs2
./inst/bin/pvfs2-ping -m /pvfs2
./inst/bin/pvfs2-ls /pvfs2/
./inst/bin/pvfs2-cp -t /usr/lib/libc.a /pvfs2/testfile1
./inst/bin/pvfs2-cp -t /pvfs2/testfile1 /tmp/testfile1-out
diff /tmp/testfile1-out /usr/lib/libc.a
cd ../../../tmp/husmann
```

- stop.sh

```
#!/bin/bash
rm -rf /tmp/husmann
killall -9 pvfs2-server
```

Their context is also quite understandable and needs no further explanation.

2.3 Eclipse Platform

Well about the configuration of eclipse is not to much to say. There have been very much changes in eclipse and especially its support of c-code since we started. So probably you'll find this part to be a lot easier than at our time. Nowadays one can even download a particular Eclipse-for-C-Developers. And I can only advise you to do so!

Therefore I will not write more about that. Only that at our time it was problematical to work with that FSM-framework precompiler! You have to try this on your own.

2.4 Message Passing Interface

The Message Passing Interface (MPI) is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today. The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few features that are language-specific. Most MPI implementations consist of a specific set of routines callable from Fortran, C, C++ or Java and from any language capable of interfacing with such routine libraries. The advantages of MPI over older message passing libraries are portability, because MPI has been implemented for almost every distributed memory architecture and speed, because each implementation is in principle optimized for the hardware on which it runs. MPI programs always work with processes, but programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multicore machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called *mpirun* or *mpiexec*.

The concept of message-passing is mainly used to distribute and concentrate informations among processes. For example a large computation is split into small jobs and spread on several cores to be computed in parallel. Subsequent all intermediate data have to be collected to compose the solution to the originally distributed problem.

Some basic commands to exchange informations are:

MPI_SEND(...) to send data from the calling process to another process specified in the attributes of this call

MPI_RECV(...) to wait for data from a specific process

MPI_Barrier(...) for synchronization (waiting) until every process reaches this command

MPI_Gather(...) tells one process to collect data from all others and tells all others to send

MPI_Reduce(...) executes a specified operation on all received data

We did use MPI a bit differently to it's intention: We used it to emulate the simultaneous access of PVFS-CF by multiple clients (processes). More about that will be found in the description of our test code which can be found in section 4.1 *Testing - MPI Test Code* at page 20. Setup will be described there, too.

We decided to use the MPICH-2 implementation of MPI because it provides PVFS support via ROMIO. In case to use the MPI Interface of PVFS you have to install MPICH-2 and - in case it is not included - ROMIO and link it to the PVFS installation. We still didn't need ROMIO, because we used MPI only to address the PVFS-CF server from multiple processors simultaneously without using the MPI interface of PVFS.

You can get MPICH2 from `http://www.mcs.anl.gov/research/projects/mpich2/`.

3 Design of PVFS-CF

As the title says, this section deals with the design of PVFS in context of its extension with the container feature, in short: **PVFS-CF**.

3.1 Excerpt from the PVFS2 Documentation

It is our believe that the following excerpt from the PVFS2 documentation (chapter 1.2.5) might be of considerable interest, therefore it is cited here:

Stateless servers and clients (1.2.5)
Parallel file systems (and more generally distributed file systems) are potentially complicated systems. As the number of entities participating in the system grows, so does the opportunity for failures. Anything that can be done to minimize the impact of such failures should be considered. NFS, for all its faults, does provide a concrete example of the advantage of removing shared state from the system. Clients can disappear, and an NFS server just keeps happily serving files to the remaining clients. In stark contrast to this are a number of example distributed file systems in place today. In order to meet certain design constraints they provide coherent caches on clients enforced via locking subsystems. As a result a client failure is a significant event requiring a complex sequence of events to recover locks and ensure that the system is in the appropriate state before operations can continue. We have decided to build PVFS2 as a stateless system and do not use locks as part of the client-server interaction. This vastly simplifies the process of recovering from failures and facilitates the use of off-the- shelf high-availability solutions for providing server failover. This does impact the semantics of the file system, but we believe that the resulting semantics are very appropriate for parallel I/O.

3.2 Overview

The main design feature of the PVFS code is the use of final state machines (FSM). They are a little complicated at first - esp. in coding - and do have quite a programming overhead, but with some time one will see the advantages:

First of all, at any moment client and server are in a well defined state! If an error occurs there is a defined error handling state to follow - which will undo all unfinished work. And if all works fine the state machine will pass into one of the next states. At the end there is always a final state which cleans up the whole working process data and frees any restricted resources. Secondly one can easily draw an outline of the states which passes a certain FSM. Now one can understand the design and basic steps within that FSM quite well. This understanding is indispensable when developing a new feature for PVFS. Working out the alternations and new states needed to be put amongst that outline suffices in defining a new design. Unfortunately the communication between these states - that's how is data transferred to the following state - is a bit mistakable; at least if you're not used to FSM programming. For understanding this: take your time!

During our design phase we fortunately found a design such that we did not have to alter the server side at all. Well apart from some common used classes like *pvfs-types.h* found in the *include* folder. Coping with our assignment we especially had to alter the client side processes *sys-create.sm* as well as the *sys-lookup.sm* which can both be found in *src/client/sysint*. Theses are both state machines thats why their extension is *.sm*. These FSM codes are a mixture of c-code and FSM structure describing code. They are refactored into standard c-code by a precompiler. The changes made to these two machines needed some enhancements of common used data structures like *object_ref* which all can be found in the above mentioned *pvfs-types.h*.

Due to some sloppiness within the original PVFS code considering the modularity of the implementation of the administration applications for PVFS there had to be carried out some modifications to these too. Changes to these programs which used the PVFS interface could point out to our failure to apply our changes in a way that would not effect any progams using the PVFS interface. But fortunately it became obvious that these modifications were rather bug fixes and did not origin in our changes but in earlier mistakes. And then all these modifications could be made in a way that users of these programs themselves wouldn't recognise them. These changes will not be described any further in this document as they - from their nature being bug fixes - were not actually part of our internship. Working with future PVFS releases it might be that they are not needed anymore. Anyway keep in mind that software might have bugs! And if you are interested in these changes check the log of folder *src/apps/admin* - all changes of this kind took place there.

Avoiding a new interface structure to create a container (like pvfs2-createcontainer e.g.) we decided to use the pvfs2-chmod program. Using the - otherwise seldom used - setgid-bit on an empty directory now marks it for further use as a container. The command `pvfs2-chmod 2777` used on an empty directory therefore creates a container with full access rights. This of course is a dirty hack - but still sufficient for evaluation purposes!

In the following sections we will go a bit deeper into our changes at *pvfs-types.h* and the means of *object_ref* first and then we will take a look at the client side create process and study our changes of that part. The lookup part will not be described any further since it's changes are quite analoge to those of the create process.

3.3 Data Types: pvfs-types.h

This file defines central data structures of the PVFS design hence it is used by both client and server. Therfore changes to it must be done carefully. Fortunately PFVS is programed quite well considering this and apart from the problems with the administration programs as mentioned in the overview the server remained unaffacted by our modifications that did only expand the structures by adding new elements. The most important though almost smallest structure defined in this file is the so called *object_ref*.

Each element in PVFS has one and only one handle which is provided by the *object_ref*. When working with any element a PVFS interface gets an *object_ref* of this element first (by sending a request containing the objects path) and from there on it does operate only on this *object_ref*. This of course is one of our problems since an *object_ref* does not contain the path any more and therefore a file within a container does not know its parent - but it gets its attributes and disk positions from there. Thats why we had to add a back link to the *object_ref* containing the container handle as well as the size/offset pair denoting where within the container the data of the file can be found (otherwise operations to get these informations on a container file handle would have been very costly - if possible):

```
typedef struct {
    PVFS_handle container_handle;
    uint64_t size; // This should be of type PVFS_size
    uint64_t offset; // This should be of type PVFS_offset
} PVFS_container_inf;
```

Finally the original *object_ref* expanded with the container stuff:

```
typedef struct {
    PVFS_handle handle; // sometimes this will be the container_handle;
    PVFS_fs_id fs_id;
    //container stuff:
    int32_t container_file;
    PVFS_container_inf cont_inf; //remember who's parent for getattr
} PVFS_object_ref;
```

As a container is a special directory where all entries refer to the contained files using key/value pairs, where the key is the filename and the value is a size/offset tuple denoting where the file is to be found within the container, we need a stucture defining such a size/offset tuple (eattr-valuepair):

```
typedef struct {
    PVFS_size size;
    PVFS_offset offset;
} PVFS_container_file_ref;
```

In a normal directory (one which is not a container) a file is refered to by an entry where the key is the filename (just as in the container) but the value contains just the handle of the file's *metafile*. You can take a look at this in figure 1 on page 14. The next section contains some more information about that.
(NB: The comments in the code snippets above do differ from those in the code!)

3.4 Creation of Files in PVFS

Before we can take a deeper look at the creation of files we will need to have a basic understanding of the data structures in PVFS. At figure 1 on page 14 we can see an example root directory pointing at it's *lost+found* subdirectory and a *README* file. Every directory points at an object of type *dirent* which encapsulates a list of all content of that directory in form of key-value pairs where the key is the name of the contained file (or directory) and the value is its PVFS handle - which is a unique number identifying any object in a PVFS file system. If it's a directory the handle points at an object of type *directory* and if it's a file it points at an object of type *metafile*. This metafile than points at a list of *datafile* objects which again defines where the actual data is found (inode, hard disk sectors).

With that basic understanding of the PVFS data structures we can now talk about the design of the create process. This process is implemented in the client's state machine: `sys-create.sm` which can be found in `src/client/sysint/`. The following quasi-copied note out of the official PVFS2-guide will desrcibe that quite intuitively. Note that the reordering of the steps needed to create a file is a very neat idea!
The create process in PVFS cosists of the following steps:

1. create a *directory* entry for the new file
2. create a *metadata* object for the new file
3. point the *directory* entry to the *metadata* object
4. create a set of *datafile* objects to hold data for the new file
5. point the *metadata* at the *datafile* objects

But thus ordered the file system would be inconsistent at certain steps. E.g. after step 1. another client could find a (new) entry whithin the directory but it would point to nothing: This would lead to an error. Therefore these steps are performed quite vice versa:

1. create a set of *datafile* objects to hold data for the new file
2. create a *metadata* object for the new file
3. point the *metadata* at the *datafile* objects
4. create a *directory* entry for the new file pointing to the *metadata* object

After step 2 neither the metadata object nor the data objects can be found by any other client than the actual working one. Therefore the file system is still consistent (looking unchanged from beyond). After step 3 that is still unchanged and after step 4 which is quasi atomic the whole new data can be seen, accessed and is fully existent.

The next two subsections will describe the create state machine in detail and the intersection we programmed to enable the container feature. But before we move on to that we should take a look at the intended advanced data structure which shall be used by PVFS-CF. In general the data structure is not changed. Only a new sub structure is added to take care of containers. Figure 2 at page 15 shows how a PVFS-CF container structure could look like.

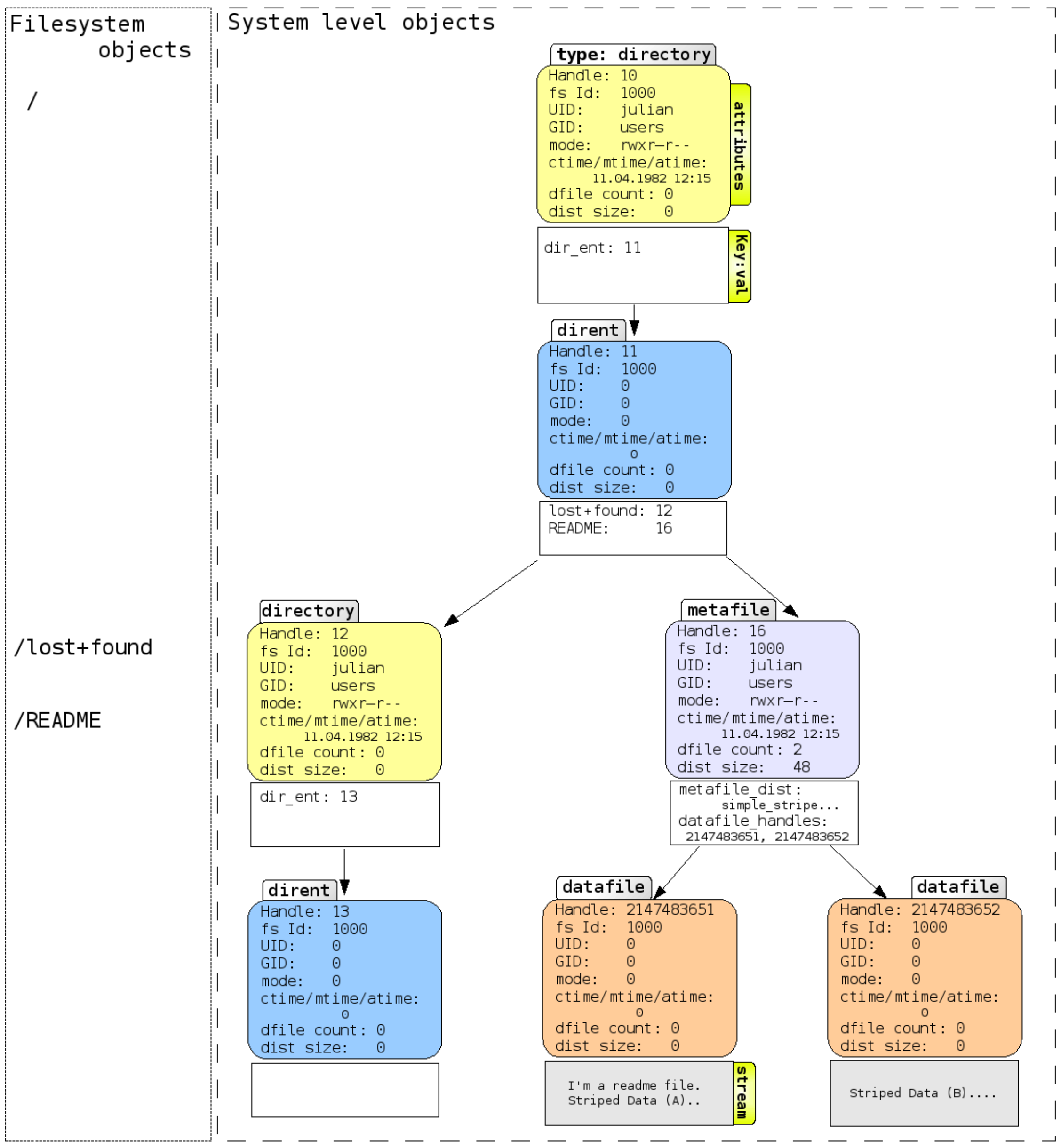

Figure 1: Original Data Structures in PVFS
Picture taken from Julian Kunkel's Master Thesis

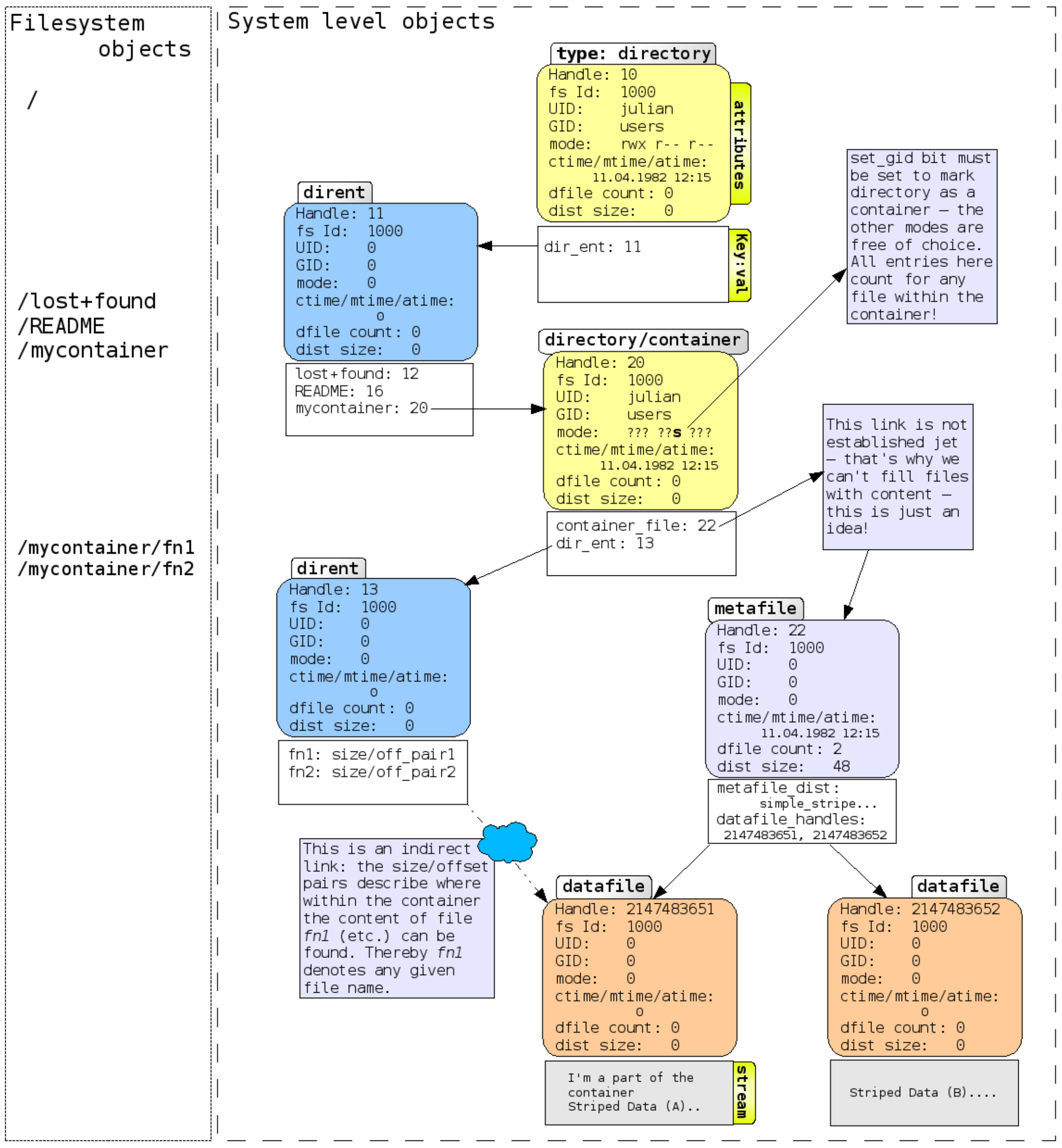

Figure 2: Advanced Data Structures in PVFS-CF
Defining a design how data shall be organised in PVFS-CF

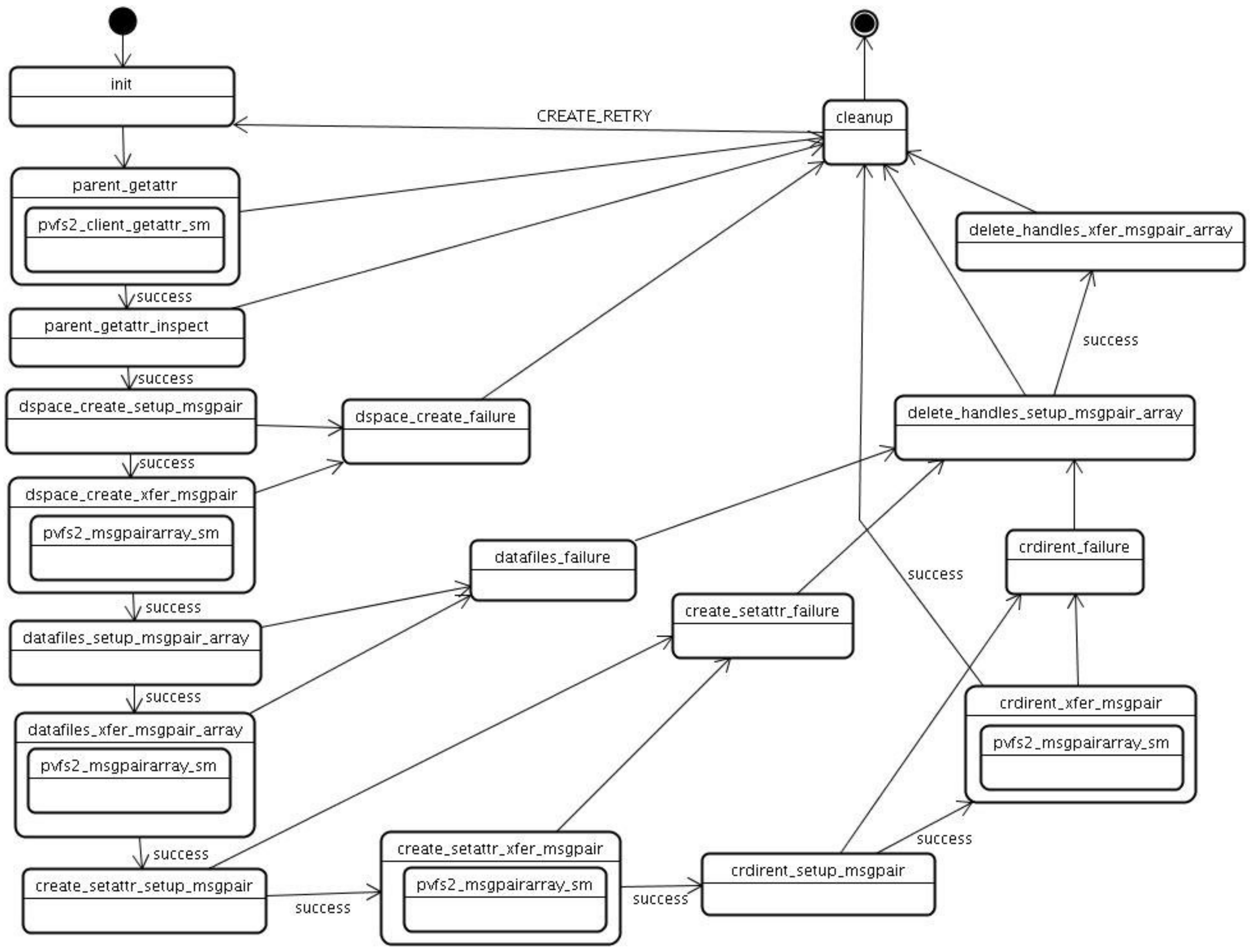

Figure 3: Create State Machine in PVFS
Sketch of the original client side process creating a file. Apart from the intersection it is quite the same as in PVFS-CF.

3.5 Original Create State Machine

Original design of state machine *sys-create.sm*. The following describes the client side process to create a file (without failures):

1.) init Initialisation of this FSM, binding resources, etc

2.) parent_getatt Read out attributes of parent directory, therefore jump in *pvfs2_client_getattr_sm*

3.) parent_getattr_inspect Inspect the attributes and continue if the creation of files is allowed

4.) dspace_create_setup_msgpair Setup msgpair to quest for storage space

5.) dspace_create_xfer_msgpair Jump in *pvfs2_msgpairarray_sm* to allocate space

6.) datafiles_setup_msgpair_array Setup msgpair containig the datafiles

7.) datafiles_xfer_msgpair_array Transfair the msgpair: jump in *pvfs2_msgpairarray_sm*

8.) create_setattr_setup_msgpair Setup msgpair for creation of metadata file and setting its attributes and pointing it to the datafiles created in the last step

9.) create_setattr_xfer_msgpair Send this msgpair: jump in *pvfs2_msgpairarray_sm*

10.) crdirent_setup_msgpair Setup msgpair for creation of direntry pointing at the new metafile

11.) crdirent_xfer_msgpair Send this msgpair: jump in *pvfs2_msgpairarray_sm*

12.) cleanup Finally cleanup: free bounded resources

3.6 Intersection CF Create State Machine

This section describes the intersection of the create state machine when creating a file within a container. The intersection takes place at point **3.) parent_getattr_inspect** of the original client side create process whose description can be found in section 3.5. The only changes at point 3. where to check the received parent attributes for the set-git bit so as to decide wether we are operating within a container. If so our code diverges from the original create process into this intersection whose outline is described below. The intersection is composed out of 4 new states (**i1.)** to **i.4)**). These new states are the major modifications that our containercode brought along. Most other modifications were dependent from our decisions how to implement this part. Only our changes to the lookup process had the same level considering their impact on our design. It was very nice that we did not have to modify any other states in this machine even though that this is due to the fact that our code does not realise the creation of files with content. That feature probably needs some more modifications to this machine.
Well, here is the list outlining this intersection:3.5.

i1.) container_get_dirdata_handle_setup_msgpair Setup msgpair to get the dirdata handle
We need this because our file needs an entry in the dirdata list of the container directory instead of it linking to its meta data which then links to its data files.

i2.) container_get_dirdata_handle_xfer_msgpair Send the above constructed msgpair:
jump in *pvfs2_msgpairarray_sm*

i3.) container_create_dirdata_setup_msgpair Setup msgpair to create a new direntry in the container, link the new file directly to the container directory and writ the size/offset pair of this file into the direntry.

i4.) container_create_dirdata_xfer_msgpair Send the above constructed msgpair:
jump in *pvfs2_msgpairarray_sm*

cleanup The new (empty) containerfile is created we can end this state machine here using the original cleanup state.

After the state *container_create_dirdata_xfer_msgpair* more intersections or container specific states will actually be needed if one really wants to copy files with content into the PVFS drive. Our internship only allows the creation of empty files within the container. The actual code of the state *container_create_dirdata_setup_msgpair* is the core of our internship and will be described in section 3.7. The full code can be found in the appendix: C.2

3.7 Create Container Direntry

The following code is cleaned of most comments as these are described in the text between the code blocks. The method is called from the state machine by the command *run container_create_dirdata_setup_msgpair* (i3). The parameters **smcb* and **js_p* are passed from the last state. Thereby **smcb* contains the status of the client state machine and **js_p* contains the job status.

```
static PINT_sm_action container_create_dirdata_setup_msgpair(
       struct PINT_smcb *smcb, job_status_s *js_p)
{
    struct PINT_client_sm *sm_p = PINT_sm_frame(smcb, PINT_FRAME_CURRENT);
    int ret = -PVFS_EINVAL;
    PINT_sm_msgpair_state *msg_p = NULL;
    PVFS_ds_keyval *cont_key = sm_p->u.create.cont_key;
    PVFS_ds_keyval *cont_val = sm_p->u.create.cont_val;

    // NB (kh): This is probably bad,
    // when putting files with content into the container:
    memset(& sm_p->u.create.cont_ref, 0, sizeof(PVFS_container_file_ref));
```

Now we have to update the *object_ref* with some container specific data:

```
    memset(& sm_p->object_ref.cont_inf, 0, sizeof(PVFS_container_inf));
    sm_p->object_ref.cont_inf.container_handle =  sm_p->object_ref.handle;
    sm_p->object_ref.handle =  sm_p->object_ref.handle;
    sm_p->object_ref.cont_inf.offset = sm_p->u.create.cont_ref.offset;
    sm_p->object_ref.cont_inf.size = sm_p->u.create.cont_ref.size;
    sm_p->object_ref.container_file = 1;
```

Secondly we have to set up the *dir_entry* message pair with the data for the new *dir_entry*. The *buffer* contains the actual contents, the *buffer_sz* defines its size. That is needed for later data retrieval.

```
    PINT_init_msgpair(sm_p, msg_p);
    /* key ist name*/
    cont_key[0].buffer = sm_p->u.create.object_name;
    cont_key[0].buffer_sz = strlen(cont_key[0].buffer) + 1;
    cont_key[0].read_sz = 0;
    /*val ist size/offset-paar definiert in conf_ref*/
    cont_val[0].buffer = & sm_p->u.create.cont_ref;
    cont_val[0].buffer_sz = sizeof(PVFS_container_file_ref);
    cont_val[0].read_sz = 0;
```

Finally the message pair is set together and its command defined, and then we continue to the next state: NB: Normally we would use *PVFS_XATTR_CREATE* here to ensure non-existence of key before creation - but we use *PVFS_XATTR_OVERRIDE* instead, to pass to the server's set-eattr.sm that we're talking about a container and therefore are not willing to check for valid namespace.

```
    PINT_SERVREQ_SETEATTR_FILL(
            sm_p->msgpair.req,
            (*sm_p->cred_p),
            sm_p->object_ref.fs_id,
            sm_p->u.create.dirdata_handle,
            PVFS_XATTR_OVERRIDE | PVFS_XATTR_CREATE,
            1,
            cont_key,
            cont_val
    );
    sm_p->msgarray = &(sm_p->msgpair);
    sm_p->msgarray_count = 1;
    sm_p->msgpair.fs_id = sm_p->object_ref.fs_id;
    sm_p->msgpair.handle = sm_p->u.create.dirdata_handle;
    sm_p->msgpair.retry_flag = PVFS_MSGPAIR_RETRY;

    ret = PINT_cached_config_map_to_server(
            &sm_p->msgpair.svr_addr,
            sm_p->msgpair.handle,
            sm_p->msgpair.fs_id);
    if (ret)
    {
        gossip_err("Failed to map meta server address\n");
        js_p->error_code = ret;
    }
    else
    {
        js_p->error_code = 0;
    }
    return SM_ACTION_COMPLETE;
}
```

4 Testing

4.1 MPI Test Code: MPI-container.c

All our tests were performed using our `MPI-container.c` code. Therefore you will find some description about that code here first.

4.1.1 Design of MPI-container.c

In our implementation (*MPI-container.c*), MPI is utilised in a slightly different way. All parallel communications are already covered by PVFS since it is a client-server based filesystem. Our goal was to design a new feature on the client side and to test this feature we needed MPI:

The messages we pass are not intended for computation, but rather simulate the interaction of many clients with the server. For this, our test scenario is based in MPI which basicly triggers the parallel access to the filesystem from various clients. These clients should initiate file operations from the commandline and measure the time needed to compute an overall performance of our newly added feature.

For example the command *mpiexec -n 16 ./MPI-container -n 10000 -s 1 -c 1 create lookup* starts the script *MPI-container* on 16 PVFS clients and tells every client to create and after that lookup (list) 10000 files using our new container feature (*-c 1*). The script calls several functions from inside the client-API which implement the functionality behind commandline operations like `~/$ touch` or `~/$ ls`.

The first part of the script executed for every client in parallel gathers some informations about the setup of the PVFS program. Then directories according to the test scenario are created. *-s 1* tells all clients to read and write into the same directory (in contrary to *-s 0*) or to create their own directory to operate in. To simulate the exaustive use of `~/$ touch <filename>` 10000 files (according to the example above) are created by calling the client side PVFS function *PVFS_sys_create(...)* with proper attributes. The use of the PFVS `~/$ ls` can be executed by calling the client side function *PVFS_sys_ref_lookup(...)*. The ability to delete files works only for those created by `~/$ touch <filename>` because our approach only focuses on the management of metadata. Thus, for empty files we are also able to delete them, calling the client side PVFS function *PVFS_sys_remove(...)* from the MPI-container program.

An additional operation of MPI is used once all operations for each client are successfully finished: Each node executing the script calculates the time of execution, these values are transfered to the root-node (how started the script on the console) by using *MPI_Gather()* and *MPI_Reduce()* to measure the overall performance, average time of execution and some best-case/worst-case estimations.

4.1.2 Usage of MPI-container.c

To compile and use MPI-container.c **MPICH-2** must be installed and up and running. The following steps describe how to do that:

1. Download and extract MPICH2 in `~/mpich2-1.0.7`
2. Prepare PVFS: cd PVFS-directory/build && make
3. cd /mpich2-1.0.7
4. ./configure –with-PVFS= /PVFS-directory/inst/ –disable-cxx –prefix= /PVFS-directory/inst/
5. make
6. make install
7. Now startMPI.sh should work!

The shell script (startMPI.sh) stops the PVFS server (if running), configures and installs the MPI server, makes MPI-container.c and thereafter MPI-container can be executed. Unfortunately startMPI.sh has to be run before each execution of MPI-container! The needed files are contained in **bundle_test.tar.gz**:

Makefile the makefile for MPI-container.c

MPI-container.c The testfile

startMPI.sh as described above

VORLAGE.sh a test script to run MPI-container multiple times with different and useful input.

The scripts will need to be run in one folder and with a defined environment variable:
`export $PVFS2HOME=<HOME OF YOUR PVFS>`.

VORLAGE.sh is german for boilerplate. So read that file, apply your changes and then you can use it. Further description will be found within these scripts.

Well lets finish this section with some details how to actually run *MPI-container.c* by itself. Since it uses MPI it has to be called using mpiexec: `mpiexec -np <processes> ./MPI-container <options>`

Thereby *processes* refers to the number of processes/clients that shall be run and communicate with the PVFS server creating or looking up files, with the *options* applying to our test code. The code runs also without mpiexec but of course then it starts only one client/process communicating with the PVFS server, simply run:
`./MPI-container <options>`

MPI-container takes the following options:

-n Number of files per client: ***1..infinite***

-c ***1***: use containerfeature (chmod 2777), ***0***: don't use it - creates a normal directory

-s ***1***: shared (all clients write to the same container/folder), ***0***: all clients write to a different one

-e, -t just some output formatting (-e better format for excel input, -t specifying a title)

create test only the creation of files

lookup test the lookup to. (needs option create because otherwise there will be nothing to look up)

4.1.3 Problems with MPI-container.c

Well altogether it worked fine but for some changes we made to the PVFS source it - unfortunately - didn't work with the original code. Therefore we could not provide any tests against the original PVFS code. Another thing is that *startMPI.sh* needs to be run before each execution. It reruns the whole PVFS server each time it is run otherwise MPI-container will fail. It's my considered opinion that this could be solved in a better way. Well see future work at page 26

4.2 Hardware: pvs-14nx

CPU Core 2 Duo E6750 (2.66 Ghz)

RAM 4 GB

DISK Seagate Barracuda 7200.10 (ST 3320620620AS)

4.3 First Test Alignment: Local Tests on pvs-14nx, one client

The first tests were performed locally on the machine `pvs-14nx`. For these tests our containercode was used and each test was performed using the container feature (uc) and compared to the results when not using the container feature (nuc). The test cases included 1k, 10k, 100k, and 1 million of files. This test alignment did not use mpiexec and therefore it used only one client. Each test case, apart from 1k, was performed five times and from the results the average was calculated (figure 4 at page 23).

There had been some strange results with the 1k tests: Unfortunately, the old code was about 42 times faster then our new code in the lookup process. That's why we run another 5 1k-tests. The results remained the same. But anyway this is not to bad since lookup of 1k files does still take only 0.42 seconds - so its still fast enough. Generally the lookup process was a little bit slower (about 10%) using the container feature than without it (figure 6 at page 24). This is a rather bad result, especially since - in normal use - lookup will be performed much more often than the creation. But therefore the creation was quite fortunate!

The creation of files was about twice as fast using the container feature (figure 5 at page 23). This is a superb result but there are reasons to question its worthiness. Since we only tested the creation of empty files, there has to be some thought on how this will perform with non-empty files. When creating one million files we archived a total time bonus of 558 seconds (about 9 minutes). When copying one million of, lets say one MB files (pictures), their total size would be a terabyte of data. It is questionable if 9 minutes of spared time are worth a lot when copying a terabyte? Well its future work to test how this will perform with non-empty files.

4.4 Second Test Alignment: Local Tests on pvs-14nx with MPI

This second test alignment (figure 7 at page 24) was performed on the same machine like the first one, but this time MPI was used. The aim of this test was to find out how well the PVFS-CF performed when called by multiple clients at the same time. Therefore for each test run R 5 clients wrote each x files into one container. So together $5x$ files. The time was measured for each client but only the slowest one was taken into account. Since just when the slowest is done everything is done.Each test run was performed 5 times and the average of the time which the slowest clients took was calculated. This result again was divided by five so that we got a fair estimation on the time x files would take. These results were compared with the results of the first test alignment (using containerfeature). The program call for the MPI part was:

```
mpiexec -np 5 ./MPI-container -n <x> -c 1 -s 1 -e -t <1..5>
```

whereby x was either 1k, 10k, 100k or one million. The results are fair enough since we got quite close results using MPI than without using it. With 10k files the MPI test was even faster. And in the end the tests were performed on a local machine. Using a real cluster surely MPI would have been quite a bit faster!

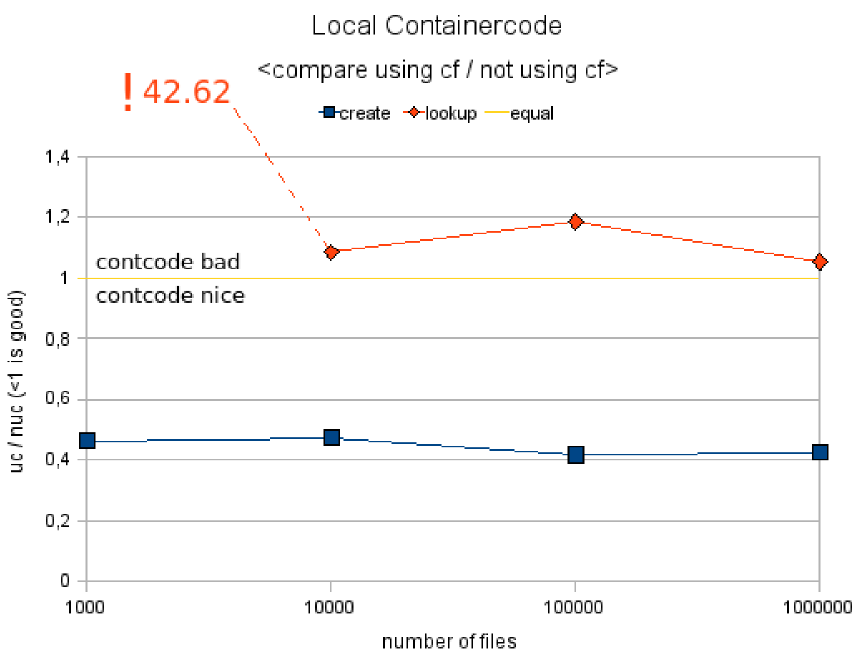

Figure 4: local comparation

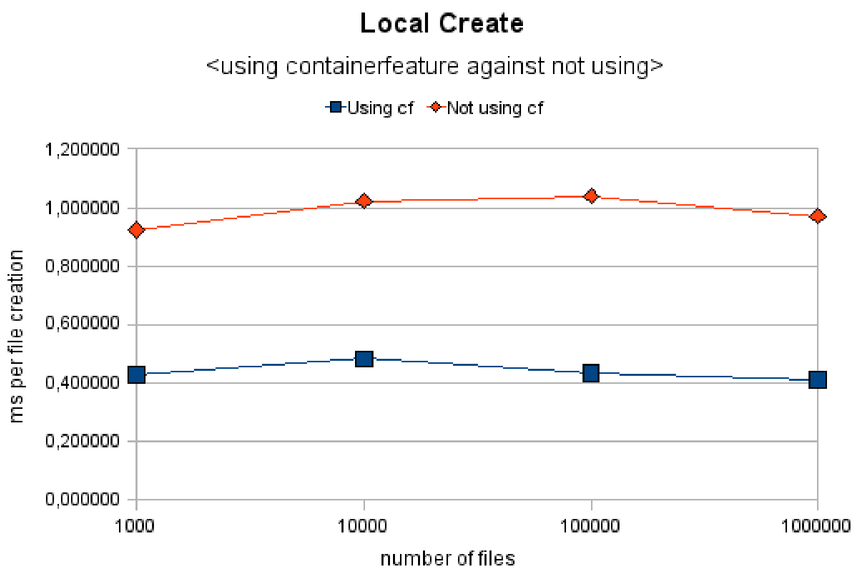

Figure 5: local comparation

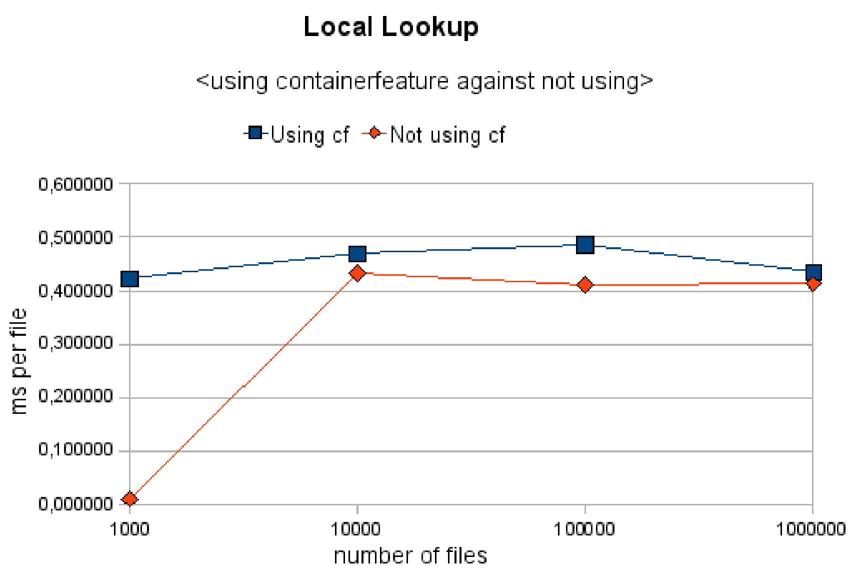

Figure 6: local comparation

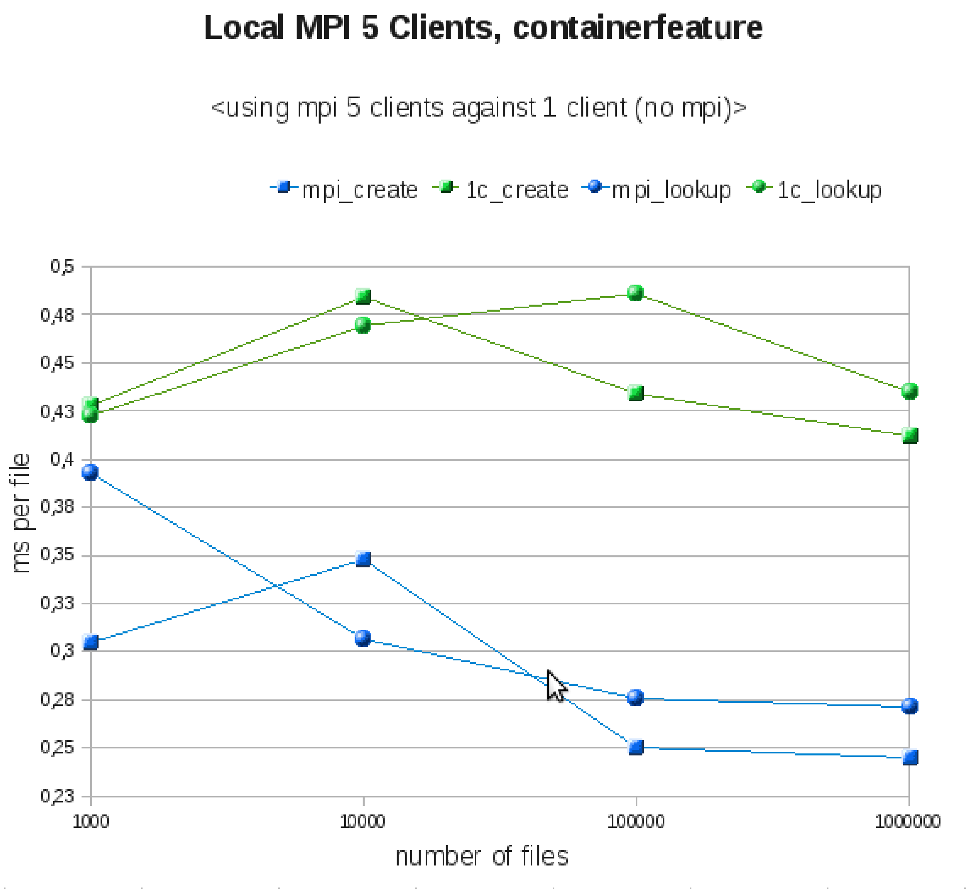

Figure 7: local MPI comparation

5 Future Work and Final Words

5.1 Future Work

At first here is a list of things that are to do:

1. Interface Make Container: *pvfs-mkcon "folder"*
2. A directory property outlining it as a container; instead of setgit bit 2777
3. The creation of files with actual content (within a container)
4. How to manage multiple clients putting data in one container?
5. Delete files from a container (maybe allow it only at the end)
6. Encapsulate the containers data within one PVFS file pointing to multiple data files
7. Retrieving data from a container,
   esp a specific file using the offset/size pair which is saved as a direntry
8. Using the PVFS cache for further enhancement
9. Refactor MPI-container.c,
   such that it works with original code as well as with the container code
10. Test it using a real cluster
11. and more..

5.1.1 pfvs-mkcon

Analogue to the clients state machine *sys-mkdir.sm* in `src\client\sysint` there should be an interface pvfs-mkcon to create a container within a PVFS directory. It should be possible to create a new empty container as well as scanning a directory (without subdirectories) and creating a container out of it putting all its contents within that new container. This functionality should have its own client state machine as well as its own system interface, named pvfs-mkcon (like make container).

5.1.2 pfvs-chmod 2777

At the moment we use pvfs-chmod 2777 to make a container out of a directory. If these bits are set PVFS-CF recognises a directory as a container. This is rather a hack than anything else. With the pfvs-mkcon system interface this should not be used anymore. Instead there should be used a direntry or anything else to mark a directory as a container. There will be need of an other property within PVFS-CF for that.

5.1.3 creation of files

At the moment our code does not allow the creation of real files (with content) within a container. This of course is not of any use apart from testing. It will probably be the most complicated feature to be build, but also the most important one to make PVFS-CF usable. Probably this needs its own state machine as well, but it should be called automatically on copying or moving files into a container. There should be no need of a system interface like `pvfs-move_file_into container`. If there should be a possibility to add data to the last (or any) file is an other question that should be decided here. The ossibility to add data to any file within a container will probably decrease efficiency seriously.

5.1.4 multiple clients

If multiple clients want to write to one container we have a serious problem since all data/ files within a container are striped and packed one after the other. Therefore it is not possible to write two files simultaneously to one container. There must either be found a solution like queueing up the files or getting the space within the container for the whole new file in advance such that another client immediately finds the new size of the container and writes its file behind that. Or this should just throw a reasonable exception. If one wants to solve this problem the first way one also needs the deletion of files within the container or otherwise when during copying the first client encounters a problem its incomplete data/ file will remain amidst the container and wasting space.

5.1.5 delete files from a container

Here are two possible ways again: Either one allows the deletion of files within a container: This will be a costly operation by all means! Or deletion within the container is prevented and only deletion of the whole container (with the possibility to restructure it's contents into a normal directory) and deletion of its last file will be allowed. The solution of the deletion problem will also be interesting for concurrent access by multiple clients

5.1.6 encapsulation of files within the container

This problem is actually much the same as the creation problem. How shall the container data be organised? Well take a look at figure 2 at page 15 to get some ideas.

5.1.7 reading data

Well, as defined for PVFS-CF any file within a container is completely specified by the meta data of the container and the file's directory entry within that container containing the file name and an offset/size pair denoting where the file data is found within that container. This is not the usual way to read a file and therefore we need a function which will return a file-like data structure constructed out of the containers meta data and the offset/size pair. Probably another state machine is needed for this. But here again another system interface should be avoided! The standard interface to read a file should just jump to another subroutine if it recognises that the file lies within a container. And finally it must return the exact same bytes as if the file would be in any normal directory!

5.1.8 cache

The PVFS code has some very efficient internal cache systems. Without any refactoring these do not function with container data. But they should! Because then a reasonable speedup for lookup and iteration through directories is expected.

5.1.9 MPI-container.c

This testing “script” must be refactored such that it operates just as well on the original code (just not calling pvfs-mkcon). But anything else must function just identically such that reasonable tests against the original code are possibly for reliably information on the enhancement the container code introduces. (And its disadvantages as well)

5.1.10 test it using a real cluster

Our MPI-container.c should work on any MPI supporting cluster but to set the whole thing (PVFS-orig and PVFS-CF) up running on a cluster is probably not trivial.

5.2 Final Words

Well, probably the best thing to do would be to check out the most up to date PVFS code and program the whole container feature from scratch again. But with a very well defined design at first!

5.2.1 project sketch

As summary I will present a little project sketch here:

1. read this document
2. install the newest stable PVFS and play with it
3. program a test script (maybe use MPI-container) which tests PVFS ability to put one million files (or so) into one directory. Throughout the process this should work on the PVFS-CF code as well as the original one.
4. try to really understand the state machines and the message passing. You will be in dire need of this understanding. Take your time.
5. study the complete creation state machines: in server and client. Outline it!
6. write pvfs-mkcon (but don't implement the creation of a container out of a full directory) to understand how internal communication works as well as the external interfaces.
7. now thing of a design and go through it multiple times until you can't find any problems any more. You can be sure of it that enough problems will arise later ;). Well where is a software designer who does not know this?
8. now you will need a project plan that allows you to restructure the design if need be and multiple test phases during your work. Find problems as early as possible!

For your project plan I will list here some assumptions on the time swallowed by some parts of the project. Well if you worked with PVFS already these won't work for you and if you haven't they might still be quite inaccurate... Esp. because some projects will integrate with others and might therefore be less consuming in the end.

1. 50 hrs reading and understanding diverse documentations
2. 80 hrs to install and understand PVFS (as user as well as a programmer) - configuring your eclipse or whatever will also take time (maybe it's easier now - c support for eclipse is by all means better nowadays)
3. 60 hrs pvfs-mkcon (for the complete thing)
4. 100 hrs for the design and its refactorings
5. 60 hrs for the creation of files (with content) within a container
6. 30-70 hrs for the deletion (depending o the supported features)
7. 40 hrs for the reading of files from the container
8. ?? hrs for the cache (I can't say anything about that)
9. 100 hrs for testing and writing test scrips and cases (some time there will be no need to be around ;)
10. And plan a minimum bonus of one month for extraneous problems!

5.2.2 Goodbye

It is in the end very difficult to understand the PVFS code and to find bugs and problems or what actually arises them! As for my opinion it is really messed up C-code and the state machines only help when you have fully understood them.

Well there we are! I'm not an English native speaker. I tried to be as near as possible to British English - but I learned English a long time ago in Ghana (no offence!). So please forgive me my mistakes. Have fun developing a final container feature for PVFS and don't forget to contact me when you're done or have any question.
And keep in mind what Monty Python once stated:

It's my considered opinion that these sheep believe they can fly.

Appendix

A Bundle test.tar.gz

A.1 startMPI.sh

```
#!/bin/bash

TESTHOME=$PWD

if [ "$PVFS_HOME" = "" ]
then
  echo Plz set PVFS_HOME to your pvfs2 folder
  echo \(where start.sh, stop.sh and pvfs2-fs.conf are found\):
  echo cd thatfolder and then type: export PVFS_HOME=\$PWD
  echo
  echo cont_code Kai\:
  echo cd ~/pvfs2/ \&\& export PVFS_HOME=\$PWD \&\& cd ../Testing/
  echo pvfs_orig Kai\:
  echo cd ~/pvfs_orig/ \&\& export PVFS_HOME=\$PWD \&\& cd ../Testing/
  echo
  exit 1
fi



echo
echo STARTING PVFS: Cleanup, build, install..
echo

# kill pvfs server
echo KILL SERVER && echo
killall -9 pvfs2-server


cd $PVFS_HOME

# free..
echo && echo FREE FOLDERS && echo

# free storage space
rm -rf /dev/shm/$USER

# free temp folder
rm -rf /tmp/$USER
mkdir /tmp/$USER

# free installation folder
rm -rf ./inst

# clear log
rm ./pvfs2-server.log
# mpi.logs und andere werden überschrieben
# build pvfs2
echo && echo BUILD PVFS && echo
make -C build -s install > make_pvfs.log

# run and config pvfs server and client
echo && echo RUN AND CONFIG SERVER && echo
./inst/sbin/pvfs2-server ./pvfs2-fs.conf -a localhost -f
./inst/sbin/pvfs2-server ./pvfs2-fs.conf -a localhost

#export PVFS2_DEBUGMASK=container

./inst/bin/pvfs2-cp pvfs2tab /pvfs2


echo "MPI -> make install"
cd ~/mpich2-1.0.7
make install > $TESTHOME/results/make_mpi.log

clear

cd $TESTHOME
echo "TEST PVFS2-Container:"
echo " make clean"
make clean > $TESTHOME/results/make_MPI-container.log

echo " make"

#make
# or #
make >> $TESTHOME/results/make_MPI-container.log


#./inst/bin/pvfs2-ls /pvfs2/
#./inst/bin/pvfs2-cp -t /usr/lib/libc.a /pvfs2/testfile
#./inst/bin/pvfs2-cp -t /pvfs2/testfile /tmp/testfile-out
#diff /tmp/testfile-out /usr/lib/libc.a
```

B Abbreviations

direntry	directory entry: an entry in an dirdata object linking to a directories content
FSM	Final State Machine
Git	An open source version control system named Git
MPI	Message Passing Interface
msgpair	message pair: a pair of orders and (optional) data to be send to the server
PVFS2	Parallel Virtual file System 2 by pvfs.org
PFVS2-CF	PVFS2 extended with the container feature evaluation (code result of this internship)
SM	State machine @see FSM
SVN	Subversion

C Code Snippets

This section reviews some parts of the code that we changed. For better readability I took the freedom to remove or change comments and debug messages.

C.1 CF Create State Machine Design

Just the code of the FSM part of `/src/client/sysint/sys-create.sm`.

```
 machine pvfs2_client_create_sm
{
    state init
    {
        run create_init;
        default => parent_getattr;
    }

    state parent_getattr
    {
        jump pvfs2_client_getattr_sm;
        success => parent_getattr_inspect;
        default => cleanup;
    }

    state parent_getattr_inspect
    {
        run create_parent_getattr_inspect;
        success => dspace_create_setup_msgpair;
        /* Andreas Beyer 28.02.2008 sm to check container_flag */
        CREATE_CONTAINER_FILE => container_get_dirdata_handle_setup_msgpair;
        default => cleanup;
    }

    state container_get_dirdata_handle_setup_msgpair
    {
        run container_get_dirdata_handle_setup_msgpair;
        success => container_get_dirdata_handle_xfer_msgpair;
        default => cleanup;
    }

    state container_get_dirdata_handle_xfer_msgpair
    {
     jump pvfs2_msgpairarray_sm;
     success => container_create_dirdata_setup_msgpair;
     default => cleanup;
    }

    state container_create_dirdata_setup_msgpair
    {
        run container_create_dirdata_setup_msgpair;
        success => container_create_dirdata_xfer_msgpair;
        default => cleanup;
    }

    state container_create_dirdata_xfer_msgpair
    {
     jump pvfs2_msgpairarray_sm;
     default => cleanup;
    }

    state dspace_create_setup_msgpair
    {
        run create_dspace_create_setup_msgpair;
        success => dspace_create_xfer_msgpair;
        default => cleanup;
    }

    state dspace_create_xfer_msgpair
    {
        jump pvfs2_msgpairarray_sm;
        success => datafiles_setup_msgpair_array;
        default => cleanup;
    }

    state datafiles_setup_msgpair_array
    {
        run create_datafiles_setup_msgpair_array;
        success => datafiles_xfer_msgpair_array;
        default => cleanup;
    }

    state datafiles_xfer_msgpair_array
    {
        jump pvfs2_msgpairarray_sm;
        success => create_setattr_setup_msgpair;
        default => datafiles_failure;
    }
```

```
state datafiles_failure
{
    run create_datafiles_failure;
    default => delete_handles_setup_msgpair_array;
}

state create_setattr_setup_msgpair
{
    run create_setattr_setup_msgpair;
    success => create_setattr_xfer_msgpair;
    default => cleanup;
}

state create_setattr_xfer_msgpair
{
    jump pvfs2_msgpairarray_sm;
    success => crdirent_setup_msgpair;
    default => create_setattr_failure;
}

state create_setattr_failure
{
    run create_setattr_failure;
    default => delete_handles_setup_msgpair_array;
}

state crdirent_setup_msgpair
{
    run create_crdirent_setup_msgpair;
    success => crdirent_xfer_msgpair;
    default => crdirent_failure;
}

state crdirent_xfer_msgpair
{
    jump pvfs2_msgpairarray_sm;
    success => cleanup;
    default => crdirent_failure;
}

state crdirent_failure
{
    run create_crdirent_failure;
    default => delete_handles_setup_msgpair_array;
}

state delete_handles_setup_msgpair_array
{
    run create_delete_handles_setup_msgpair_array;
    success => delete_handles_xfer_msgpair_array;
    default => cleanup;
}
```

```
    state delete_handles_xfer_msgpair_array
    {
        jump pvfs2_msgpairarray_sm;
        default => cleanup;
    }

    state cleanup
    {
        run create_cleanup;
        CREATE_RETRY => init;
        default => terminate;
    }
}
```

C.2 Creation of containerspecific direntry

This c-code (textitcontainer_create_dirdata_setup_msgpair is run by the eqally named state within the CF create state machine. It is somewhat the most important code part of this internship.

```
 static PINT_sm_action container_create_dirdata_setup_msgpair(
struct PINT_smcb *smcb, job_status_s *js_p)
{
    if (js_p->error_code == -1073741826)
      gossip_debug(GOSSIP_CONTAINER_DEBUG,
##"   js_p->error_code = \"File not found\"\n");
    else
    struct PINT_client_sm *sm_p = PINT_sm_frame(smcb, PINT_FRAME_CURRENT);
    int ret = -PVFS_EINVAL;
    PINT_sm_msgpair_state *msg_p = NULL;

    PVFS_ds_keyval * cont_key = sm_p->u.create.cont_key;
    PVFS_ds_keyval * cont_val = sm_p->u.create.cont_val;

    memset(& sm_p->u.create.cont_ref, 0, sizeof(PVFS_container_file_ref));

    // Allocate memory for cont_inf and fill it with:
    // container_handle, size und offset)

    memset(& sm_p->object_ref.cont_inf, 0, sizeof(PVFS_container_inf));
    sm_p->object_ref.cont_inf.container_handle =  sm_p->object_ref.handle;
    sm_p->object_ref.handle =  sm_p->object_ref.handle;
    sm_p->object_ref.cont_inf.offset = sm_p->u.create.cont_ref.offset;
    sm_p->object_ref.cont_inf.size = sm_p->u.create.cont_ref.size;
    sm_p->object_ref.container_file = 1;

    PINT_init_msgpair(sm_p, msg_p);

    /* key ist name*/
    cont_key[0].buffer = sm_p->u.create.object_name;
    cont_key[0].buffer_sz = strlen(cont_key[0].buffer) + 1;
    cont_key[0].read_sz = 0;
    /*val ist size/offset-paar definiert in conf_ref*/
    cont_val[0].buffer = & sm_p->u.create.cont_ref;
    cont_val[0].buffer_sz = sizeof(PVFS_container_file_ref);
    cont_val[0].read_sz = 0;

/* normally we would use PVFS_XATTR_CREATE here to ensure
 * non-existance of key before creation - but we
 * use PVFS_XATTR_OVERRIDE instead, to pass to the server's
 * set-eattr.sm that we're talking about a container and
 * therefore are not willing to check for valid namespace.*/
    PINT_SERVREQ_SETEATTR_FILL(
            sm_p->msgpair.req,
            (*sm_p->cred_p),
            sm_p->object_ref.fs_id,
            sm_p->u.create.dirdata_handle,
            PVFS_XATTR_OVERRIDE | PVFS_XATTR_CREATE,
            1,
            cont_key,
            cont_val
    );
    sm_p->msgarray = &(sm_p->msgpair);
    sm_p->msgarray_count = 1;
    sm_p->msgpair.fs_id = sm_p->object_ref.fs_id;
    sm_p->msgpair.handle = sm_p->u.create.dirdata_handle;
    sm_p->msgpair.retry_flag = PVFS_MSGPAIR_RETRY;
    /* NOTE: no comp_fn needed. */

    ret = PINT_cached_config_map_to_server(
            &sm_p->msgpair.svr_addr,
            sm_p->msgpair.handle,
            sm_p->msgpair.fs_id);
    if (ret)
    {
        gossip_err("Failed to map meta server address\n");
        js_p->error_code = ret;
    }
    else
    {
        js_p->error_code = 0;
    }
    return SM_ACTION_COMPLETE;
}
```