OrangeFS

Hochleistungs-Ein-/Ausgabe

Michael Kuhn

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

2017-05-12

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

E/A-Schichten

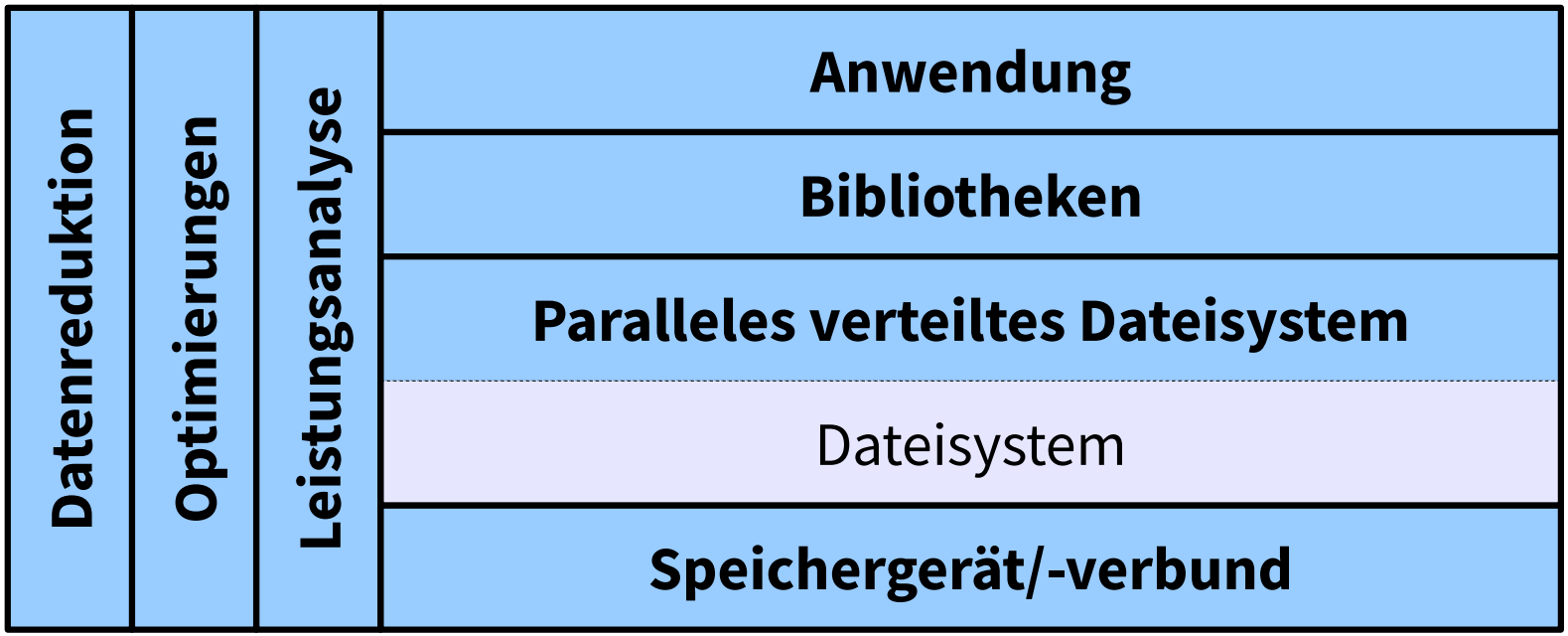


Abbildung: E/A-Schichten

Überblick

- Weiterentwicklung von PVFS
  - Entwicklung durch Clemson University, Argonne National Laboratory und Omnibond
- Open Source (LGPL)
  - $\geq$ 250.000 Zeilen Code
- Läuft komplett im User-Space
- Beliebt für Forschungsprojekte
  - Deutlich einfacherer Einstieg als Lustre

Historie

- 1993: PVFS Version 0 als NASA-Projekt
- 1994: PVFS Version 1 mit TCP-Unterstützung
- 1997: Veröffentlichung als Open Source
- 2003: Veröffentlichung von PVFS Version 2
- 2008: Start von OrangeFS als Entwicklungszweig
- 2010: OrangeFS ersetzt PVFS als Hauptversion
- 2011: Unterstützung für Windows-Clients
- 2016: Kernel-Client in Linux 4.6 hinzugefügt

Historie…

- 2.8.5: Server-zu-Server-Kommunikation, Replikation für unveränderbare Dateien, SSD-Unterstützung für Metadaten
- 2.8.6: Direct-Interface-Bibliotheken, Unterstützung für WebDAV und S3
- 2.8.8: Unterstützung für Hadoop MapReduce
- 2.9: Neue Sicherheitsmechanismen, verteilte Metadaten für Verzeichniseinträge [4]
- 2.9.2: Unterstützung für “lazy atime”
- 2.9.5: Unterstützung für LMDB-Metadaten-Backend
- 2.9.6: Readahead-Caching im Client

- Die SSD-Unterstützung wird dadurch realisiert, dass die Pfade für Daten und Metadaten separat konfiguriert werden können
- Bis Version 2.9 wurden alle Einträge eines Verzeichnisses von einem Metadatenserver verwaltet
  - Die eigentlichen Daten waren schon immer verteilt

Funktionalität

- Sehr gute MPI-IO-Unterstützung
  - Natives Backend in ROMIO
  - Entwicklergruppe um MPICH und ROMIO
- Unterstützung für POSIX-Schnittstellen
  - Direct-Interface-Bibliotheken
  - FUSE-Dateisystem
    - Primär für OS X
  - Optionales Kernelmodul
    - Seit Linux 4.6 im Kernel enthalten

- OrangeFS hat unter anderem Unterstützung für kollektive und nicht-zusammenhängende Ein-/Ausgabe

Installation

- Sehr einfache Benutzung und Installation

```
1  $ {apt,dnf,yum} install bison flex libdb-dev[el]
       ↪ libattr-dev[el]
2
3  $ wget .../orangefs-x.y.z.tar.gz
4  $ tar xf orangefs-x.y.z.tar.gz
5  $ cd orangefs-x.y.z
6
7  $ ./configure --prefix=${PREFIX} --enable-shared
8  $ make --jobs=$(nproc)
9  $ make install
```

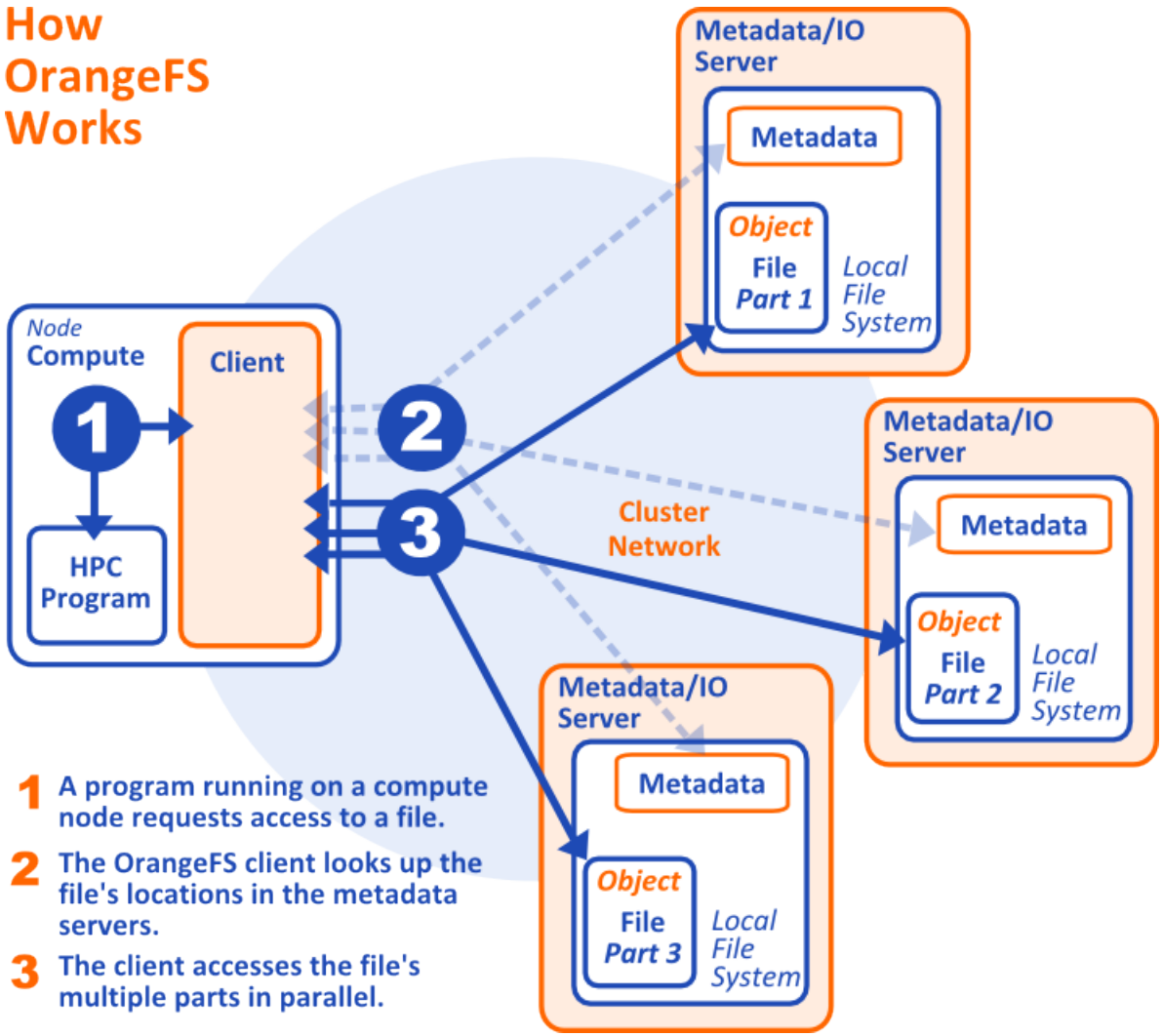
Konfiguration

```
$ pvfs2-genconfig --quiet --protocol tcp,ib \
                  --ioservers west{1-10} \
                  --metaservers west{1-10} \
                  --storage /tmp/orangefs-${USER} \
                  --metadata /tmp/orangefs-${USER} \
                  --logfile /tmp/orangefs-${USER}.log \
                  orangefs.conf
```

```
$ clush -w west[1-10] pvfs2-server --mkfs orangefs.conf
$ clush -w west[1-10] pvfs2-server orangefs.conf
$ clush -w west[1-10] pvfs2-server --rmfs orangefs.conf
```

- OrangeFS wird mit folgenden Optionen konfiguriert:
  - Unterstützung für TCP und InfiniBand
  - Auf west[1-10] läuft jeweils ein Daten- und ein Metadatenserver
  - Daten und Metadaten werden im selben Verzeichnis gespeichert
    - Das Verzeichnis kann sich in einem beliebigen lokalen Dateisystem befinden
  - Es wird eine Protokolldatei geschrieben
- `clush` erlaubt das Ausführen von Befehlen auf mehreren Knoten
  - Mit dem `mkfs`-Parameter wird das Verzeichnis für Daten und Metadaten initialisiert
  - Mit dem `rmfs`-Parameter wird das Verzeichnis wieder gelöscht

Funktionsweise [2]



- Die Metadaten einer Datei befinden sich auf einem einzelnen Metadatenserver
  - Unter Umständen müssen aber mehrere Metadatenserver kontaktiert werden, um die Datei zu finden

Funktionsweise…

- OrangeFS ist objekt-basiert
  - Vier Hauptobjekttypen: Datafile, Metafile, Directory, Symlink
- Dateien/Verzeichnisse bestehen aus mindestens zwei Objekten
  - Eines für Metadaten und die restlichen für Daten
- Server arbeiten mit Objekten, Bytestreams und Key-Value-Paaren
  - Objekte haben eindeutige Handles
  - Bytestreams für Daten
  - Key-Value-Paare für Attribute und Metadaten
  - Objekte nutzen Bytestreams und/oder Key-Value-Paare

Funktionsweise…

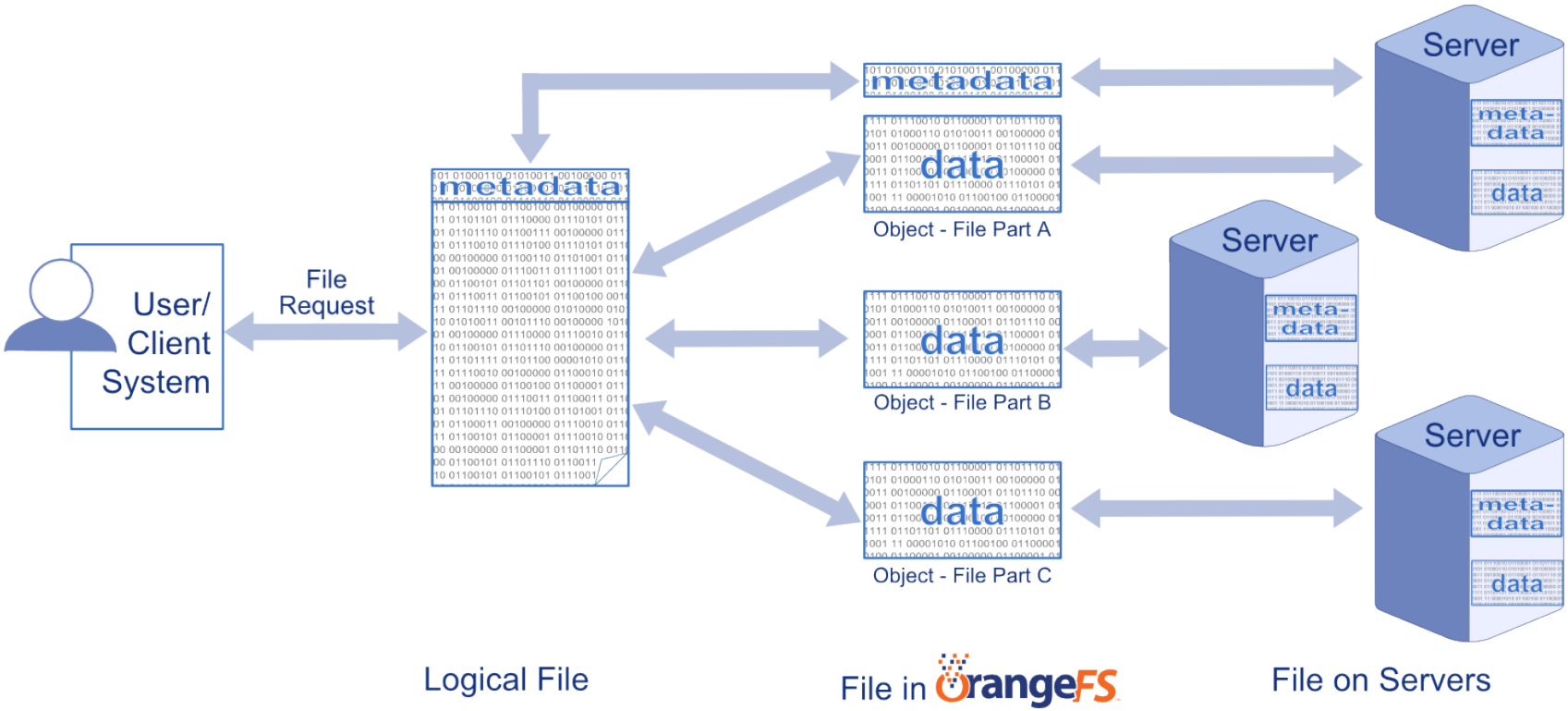
- Key-Value-Paare werden mit der Berkeley DB verwaltet
  - Alternativ mit LMDB
- Bytestreams werden im lokalen Dateisystem gespeichert

```
$ ls -lh
drwxr-xr-x 4 wr wr 4.0K Jun 25 20:45 3b3f08ea
-rw------- 1 wr wr 8.0K Jun 25 20:45 collections.db
-rw------- 1 wr wr 8.0K Jun 25 20:45 storage_attributes.db
$ ls -lh 3b3f08ea/
drwxr-xr-x 66 wr wr 4.0K Jun 25 20:45 bstreams
-rw-------  1 wr wr 8.0K Jun 25 20:45 collection_attributes.db
-rw-------  1 wr wr 992K Jun 28 19:42 dataspace_attributes.db
-rw-------  1 wr wr   16 Jun 25 20:45 __db.001
-rw-------  1 wr wr 228K Jun 27 18:38 keyval.db
drwxr-xr-x  2 wr wr 4.0K Jun 28 19:42 stranded-bstreams
$ ls -lh 3b3f08ea/bstreams/00000021/
-rw------- 1 wr wr 416M Jun 28 22:48 466666666665515.bstream
```

Funktionsweise…

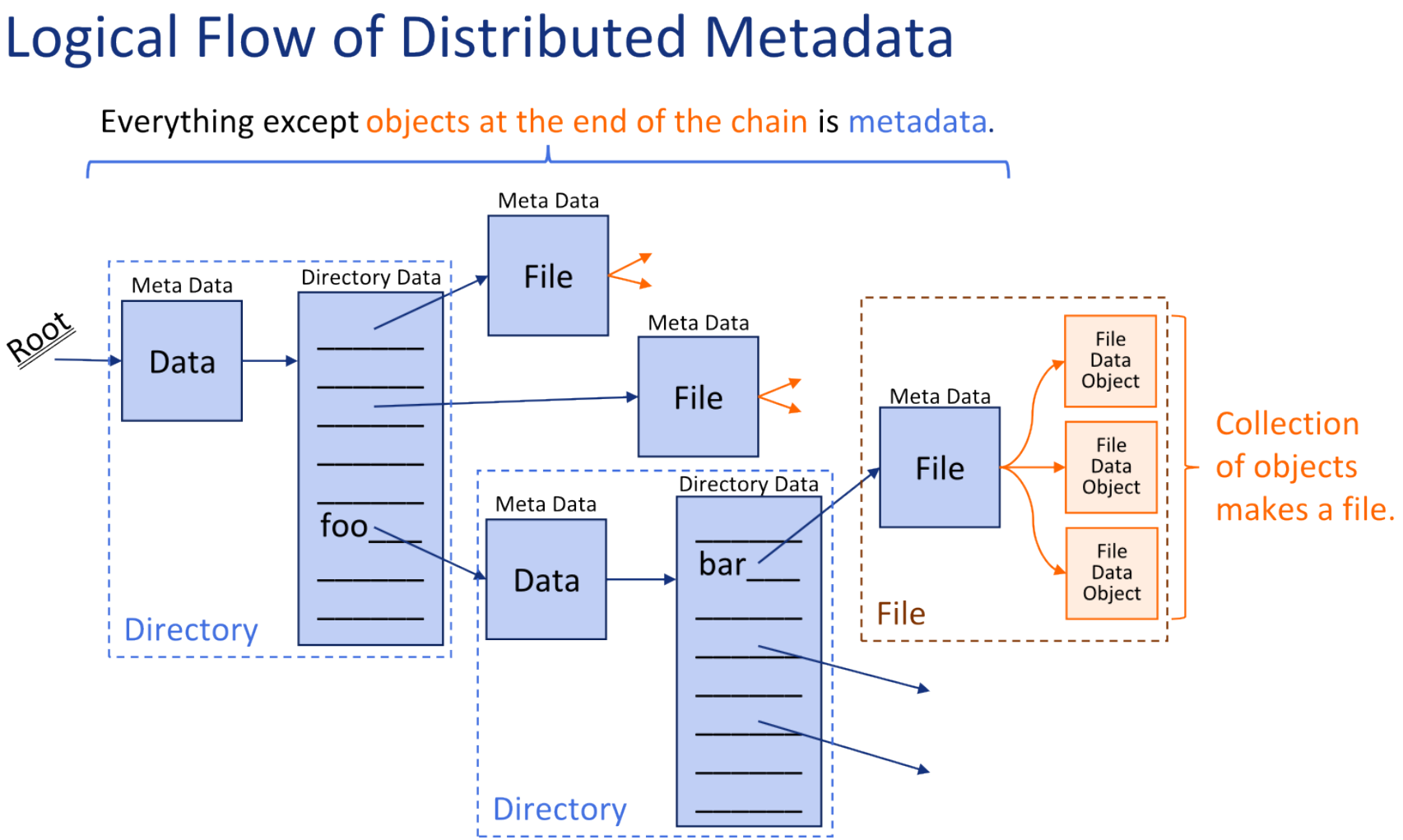
- Metafiles repräsentieren Dateien
  - Speichern Metadaten wie Eigentümer und Berechtigungen
  - Enthalten außerdem Handles mehrerer Datafiles (inklusive deren Verteilung)
  - Größe wird nicht explizit gespeichert
- Datafiles repräsentieren Daten
  - Verteilt über alle Server
  - Keine Metadaten, nur eigentliche Dateiinhalte
- Directories repräsentieren Verzeichnisse
  - Speichern Metadaten wie Eigentümer und Berechtigungen
  - Außerdem Handles von Directory-Data-Objekten
- Symlinks repräsentieren symbolische Verweise

Funktionsweise... [2]



- Die logische Datei besteht aus Metadaten und Daten
- Die physikalische Datei besteht aus einem Metafile und drei Datafiles
  - Der erste Server enthält sowohl das Metafile als auch das erste Datafile
  - Die restlichen Server enthalten jeweils ein Datafile

Funktionsweise... [2]



- Um eine Datei zu finden, müssen möglicherweise mehrere Metadatenserver kontaktiert werden
  - Zuerst wird das Wurzelverzeichnis durchsucht, dann die nächste Pfadkomponente etc.

Schnittstellen

- Zugriff wird in drei Ebenen eingeteilt
  - C-Bibliothek: `fopen`, `fread` etc.
  - POSIX-Bibliothek: `open`, `read` etc.
    - Funktionieren mit OrangeFS- und POSIX-Dateisystemen
  - System-Call-Bibliothek: `pvfs_open`, `pvfs_read` etc.
    - Funktioniert nur mit OrangeFS-Dateisystemen
- Das Direct Interface kann explizit oder implizit genutzt werden
  - Dafür stehen vier Bibliotheken zur Verfügung: `liborangefsposix`, `liborangefs`, `libofs` und `libpvfs2`

- Explizite Nutzung bedeutet, dass die Anwendungen gegen die entsprechenden Bibliotheken gelinkt werden
- Implizite Nutzung bedeutet, dass die entsprechenden Bibliotheken über den Preload-Mechanismus geladen werden

Schnittstellen...

- `liborangefsposix`
  - Deckt die C- und POSIX-Bibliotheken ab
  - Eine Zusammenfassung der anderen Bibliotheken
- `liborangefs`
  - Deckt die System-Call-Bibliothek ab
  - Enthält POSIX-PVFS-Funktionen
    - `pvfs_open`, `pvfs_read`, `pvfs_write`, ...
    - Verhalten sich wie die entsprechenden POSIX-Funktionen

Schnittstellen...

- `libofs`
  - Deckt die C- und POSIX-Bibliotheken ab
  - Enthält Funktionen aus POSIX (inkl. stdio), FTS und glob
    - `open`, `read`, `fopen`, `fread`, `fts_open`, `fts_read`, ...
    - Verhalten sich genauso wie Originalfunktionen
- `libpvfs2`
  - Deckt die System-Call-Bibliothek ab
  - Enthält das native System Interface
    - `PVFS_sys_lookup`, `PVFS_sys_getattr`, ...
  - Nicht für direkte Benutzung gedacht
    - Basis für entwicklerfreundliche APIs

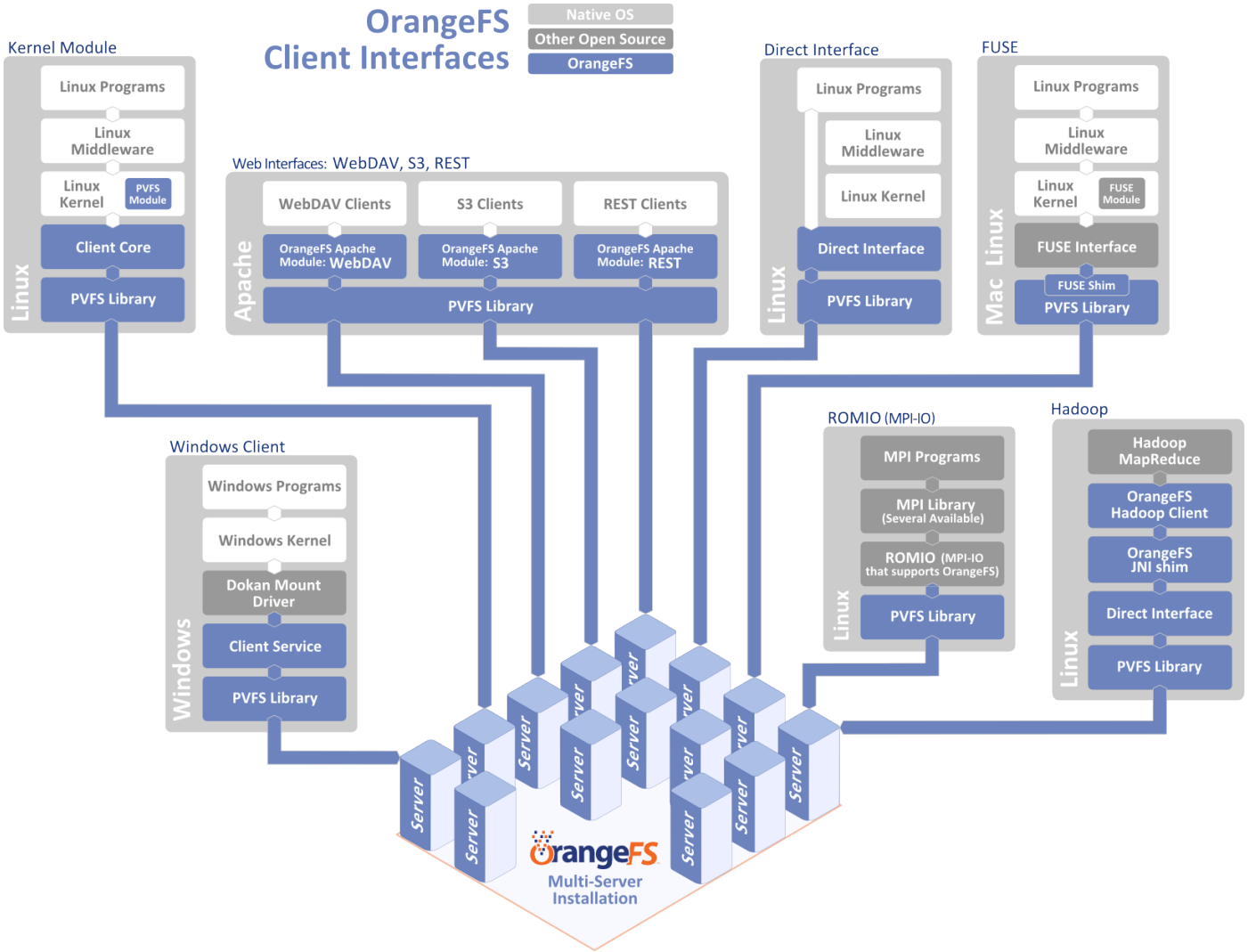
Schnittstellen...

- Anwendungen können direkt gegen entsprechende Bibliotheken gelinkt werden
- Alternativ Preloading von Bibliotheken

```
1 $ gcc -o posix_app -L${PREFIX}/lib posix_app.c -lorangefsposix
2 $ ./posix_app
3 $ gcc -o orange_app -L${PREFIX}/lib orange_app.c -lorangefs
4 $ ./orange_app
```

```
1 $ export LD_LIBRARY_PATH=${PREFIX}/lib
2 $ export LD_PRELOAD=${PREFIX}/lib/libofs.so
3 $ ./legacy_app
```

Schnittstellen… [2]



Schnittstellen…

- POSIX-Schnittstellen erlauben auch Zugriff auf OrangeFS-Funktionalität
- Geregelt über spezielle Flags beim `open`-Aufruf
  - Für `pvfs_open` und `open` aber nicht für `fopen`
  - Beispiele:
    Verteilungsfunktion (`PVFS_HINT_DISTRIBUTION_NAME`),
    Anzahl der Datafiles (`PVFS_HINT_DFILE_COUNT_NAME`),
    Caching (`PVFS_HINT_CACHE_NAME`) etc.
- Meistens nur Auswirkungen beim Erstellen

Schnittstellen...

```
1  int fd;
2  PVFS_hint myhint = NULL;
3  int layout = PVFS_SYS_LAYOUT_RANDOM;
4
5  PVFS_hint_add(&myhint, PVFS_HINT_LAYOUT_NAME, sizeof(layout),
       ↪ &layout);
6
7  fd = pvfs_open("/path/to/file", O_CREAT | O_RDWR | O_TRUNC |
       ↪ O_HINTS, 0644, myhint);
```

Listing 1: OrangeFS-spezifische Hinweise [3]

Verteilungsfunktionen

- Unterstützung für vier Verteilungsfunktionen
- `basic_dist`
  - Speicherung in einem Datafile
  - Eventuell nützlich für kleine Dateien
  - Keine Parameter einstellbar
- `simple_stripe` (Standard)
  - Round Robin mit fester Streifenbreite
  - Entspricht der Verteilungsfunktion in Lustre
  - Parameter:
    - `strip_size`: Streifenbreite (64 KiB)

Verteilungsfunktionen...

- `twod_stripe`
  - Verteilung nach 2D-Muster über Gruppen von Datafiles
  - Erlaubt Einteilung von Servern in Gruppen
  - Parameter:
    - `num_groups`: Anzahl Datafile-Gruppen
    - `strip_size`: Streifenbreite
    - `group_strip_factor`: Anzahl Streifen pro Server und Gruppe bevor nächste Gruppe benutzt wird
- `varstrip_dist`
  - Round Robin mit variabler Streifenbreite
  - Erlaubt Anpassung an spezielle Datenformate
  - Parameter:
    - `strips`: Streifenbreiten pro Server
    - Beispiel: `0:32K; 1:128K; 2:64K; 3:128K;`

- Die `twod_stripe`-Verteilungsfunktion kann genutzt werden, um die Anzahl der zu kontaktierenden Server einzuschränken
  - Beispiel: n Clients greifen gemeinsam auf eine Datei der Größe m zu, dabei ist jeder Client für m/n zusammenhängende Bytes verantwortlich
  - Bei entsprechender Wahl der Parameter kann dafür gesorgt werden, dass jeder Client nur auf eine Untermenge der verfügbaren Server zugreift, insgesamt aber auf alle Server zugegriffen wird
- Die `varstrip_dist`-Verteilungsfunktion kann beispielsweise genutzt werden, um sicherzustellen, dass logisch zusammenhängende Teile einer Datei auf einem Datenserver gespeichert werden

Verteilungsfunktionen…

```
1  PVFS_sys_dist* new_dist;
2  PVFS_size strip_size = 131072;
3
4  new_dist = PVFS_sys_dist_lookup("simple_stripe");
5
6  PVFS_sys_dist_setparam(new_dist, "strip_size", &strip_size);
7  PVFS_sys_create(entry_name, parent_ref, attr, credentials,
       ↪ new_dist, &resp_create, NULL, hints);
```

Listing 2: Verteilungsfunktion setzen [3]

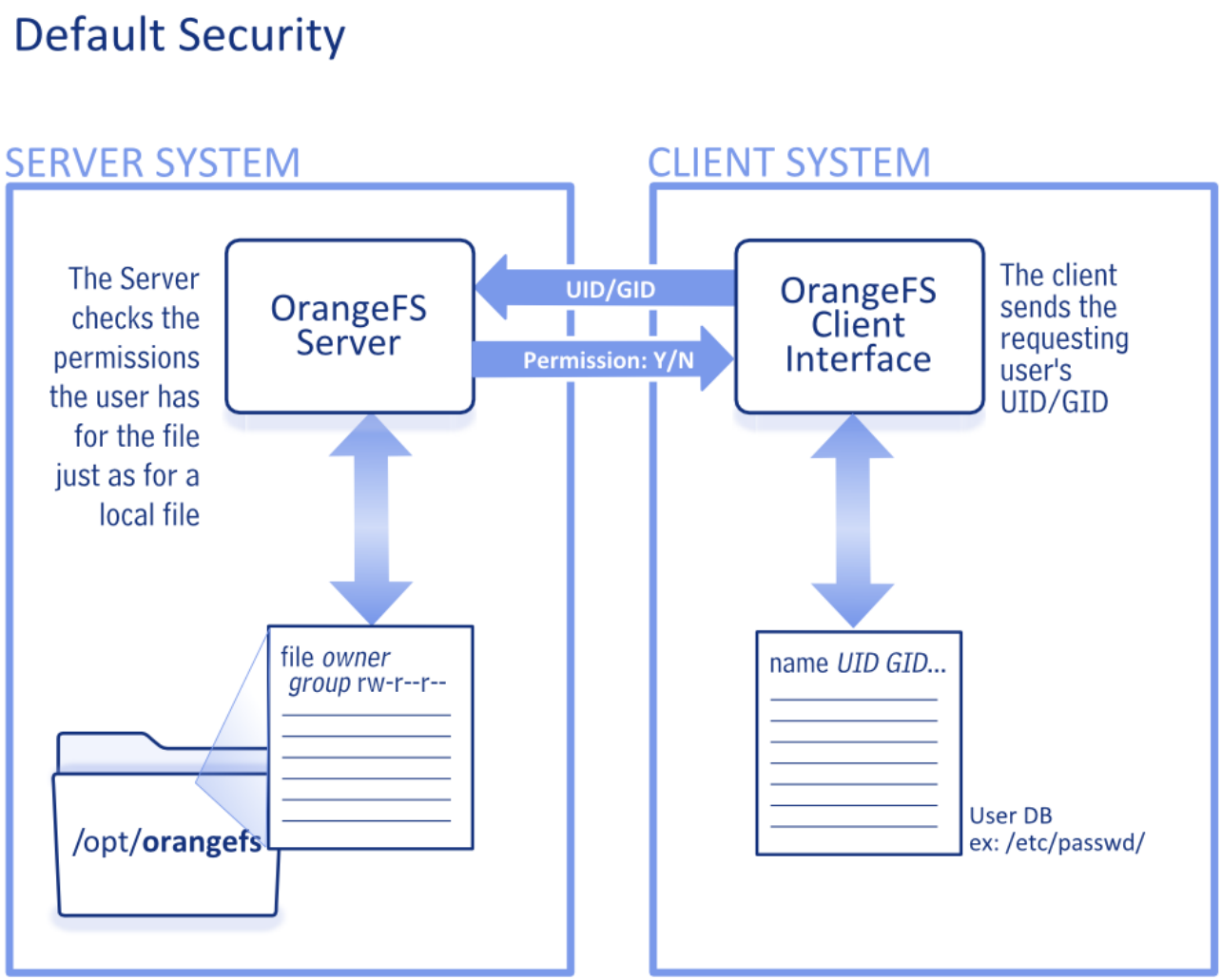
Verteilungsfunktionen…

- Zusätzlich Unterstützung für vier Layouts
  - Legen die Reihenfolge der Server fest
- `PVFS_SYS_LAYOUT_NONE`
  - Reihenfolge wie in der Konfigurationsdatei angegeben
  - Start beim ersten Server
- `PVFS_SYS_LAYOUT_ROUND_ROBIN` (Standard)
  - Reihenfolge wie in der Konfigurationsdatei angegeben
  - Start bei einem zufälligen Server
- `PVFS_SYS_LAYOUT_RANDOM`
  - Reihenfolge der Server ist zufällig (ohne Wiederholung)
- `PVFS_SYS_LAYOUT_LIST`
  - Reihenfolge der Server ist wie explizit angegeben

Sicherheit

- Unterstützung für drei Sicherheitsmechanismen
  - Standard, schlüssel-basiert und zertifikat-basiert mit LDAP
  - Alle Mechanismen arbeiten mit Timeouts
- Zugriffskontrolle wird über Credentials geregelt
  - Standardmäßig senden Clients Credentials
  - Server überprüft Berechtigungen auf Basis der Credentials

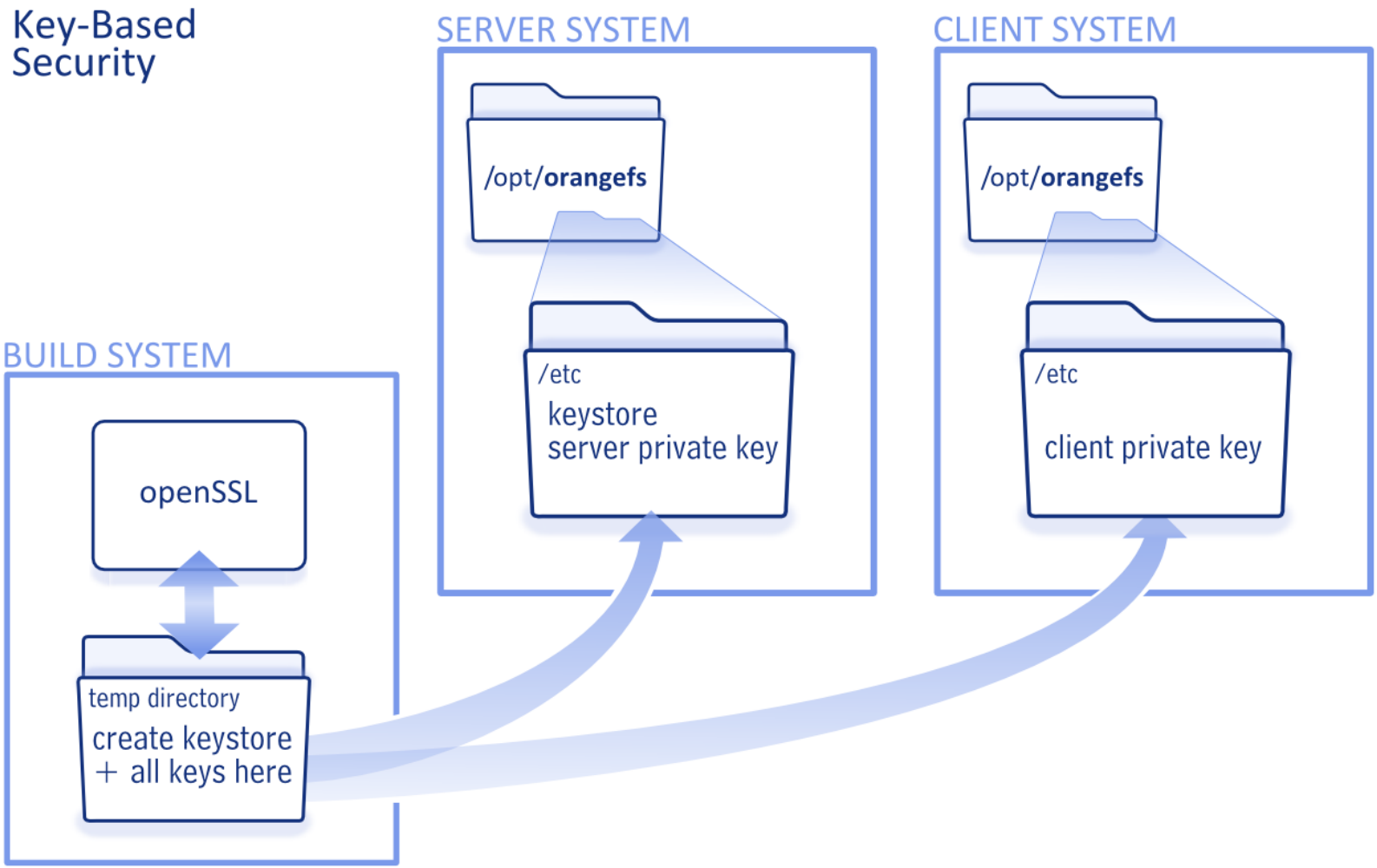
Sicherheit… [2]



Sicherheit…

- Benötigt keinerlei Konfiguration
  - Einfache Installation und Konfiguration
  - Gut geeignet für Evaluationen und Tests
- Hohe Leistung
  - Keine zusätzlichen Abfragen und Dienste notwendig
- Keine hohe Sicherheit
  - Clients können beliebige Credentials senden
- Auch bei anderen Dateisystemen häufig anzutreffen
  - Lustre arbeitet in der Standardkonfiguration ähnlich

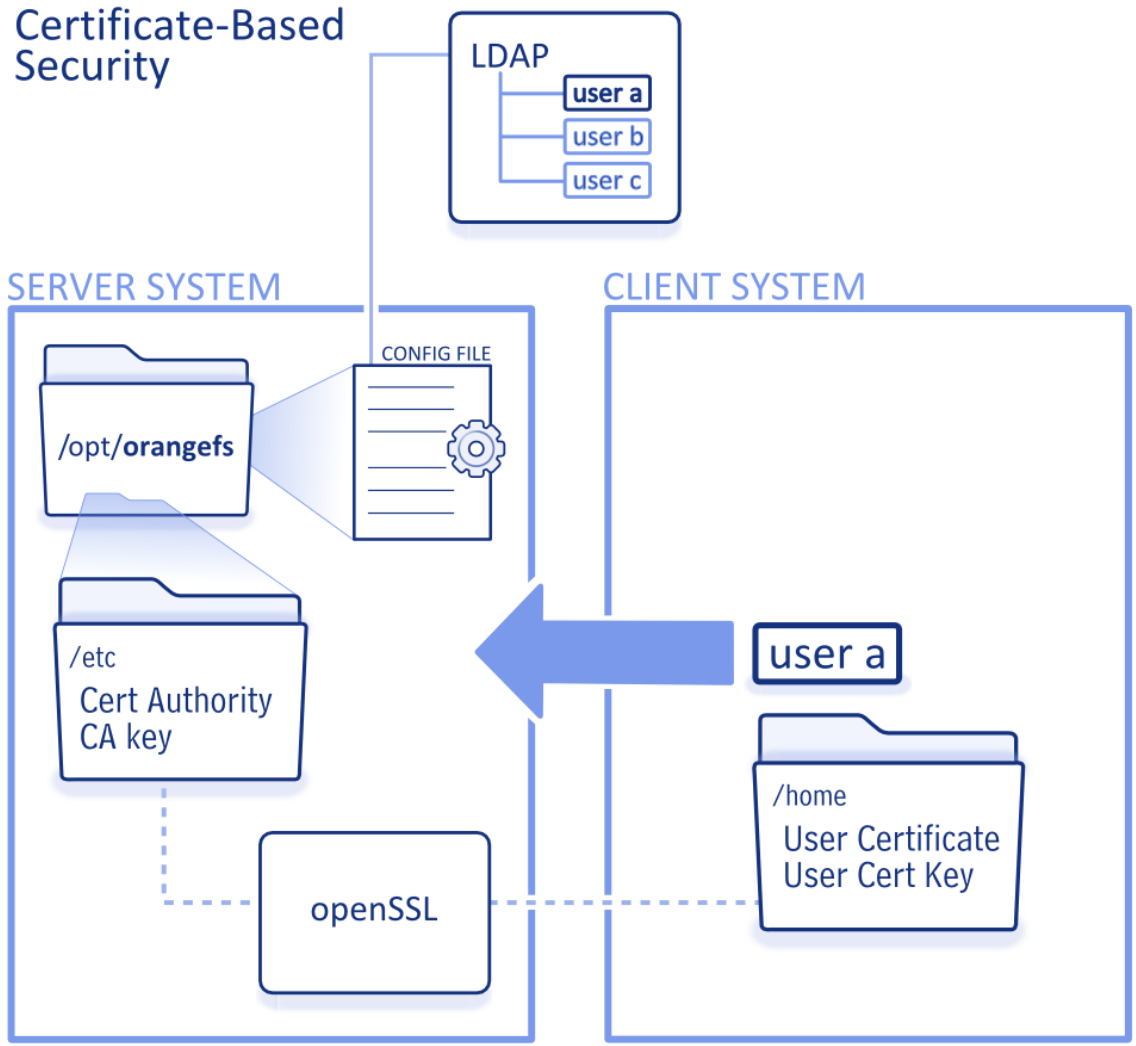
Sicherheit… [2]



Sicherheit…

- Benutzt private und öffentliche Schlüssel zur Authentifizierung
  - Jeder Server und Client hat eigenes Schlüsselpaar
  - Server nutzen einen Schlüsselspeicher, der alle öffentlichen Schlüssel enthält
- Credentials werden signiert
  - Server prüft zusätzlich Signatur
- Schlüsselspeicher ist schwierig zu handhaben
  - Muss bei Änderungen neu generiert und verteilt werden
  - Server müssen neugestartet werden

Sicherheit… [2]



Sicherheit…

- Alle Server teilen sich eine Zertifizierungsstelle
  - Zertifizierungsstelle stellt weitere Zertifikate aus
- Jeder Benutzer hat eigenes Zertifikat
  - Wird im Home-Verzeichnis gespeichert
  - Zuordnung von Zertifikat zu Benutzer- und Gruppen-ID mit Hilfe von LDAP
- Höherer Aufwand als andere Mechanismen
  - LDAP-Installation falls noch nicht vorhanden
  - Komplexere und zusätzliche Schritte notwendig
  - Allerdings keine Server-Neustarts notwendig

Schichten

- Job Manager
  - Operationen bestehen aus mehreren Schritten (Jobs)
  - Koordination aller Jobs und Thread-Verwaltung
- Flows
  - Bezeichnet abstrakte Datenflüsse
  - Koordination mit anderen Schichten
- BMI
  - Abstraktion des Netzwerks (Buffered Message Interface)
  - Verwaltung der Netzwerkkommunikation
- Trove
  - Abstraktion der Speichers
  - Verwaltung von Key-Value-Paaren und Bytestreams

Operationen

- Zwei Varianten von allen Operationen
  - `PVFS_sys_op` und `PVFS_isys_op`
  - Synchron und asynchron
- Eigentliche Funktionalität asynchron implementiert
  - Synchrone Variante ruft asynchrone auf
  - `PVFS_sys_wait` und `PINT_sys_release`
- Vielzahl an `structs` und `unions`
  - Generische Datenstrukturen für alle Operationen

Operationen... [1]

```
1  PVFS_error PVFS_isys_create (...)
2  {
3      PINT_smcb *smcb = NULL;
4      PINT_client_sm *sm_p = NULL;
5      ...
6      PINT_smcb_alloc(&smcb, PVFS_SYS_CREATE, ...);
7      sm_p = PINT_sm_frame(smcb, PINT_FRAME_CURRENT);
8      ...
9      sm_p->u.create.object_name = object_name;
10     ...
11     sm_p->parent_ref = parent_ref;
12     sm_p->object_ref = parent_ref;
13     ...
14     return PINT_client_state_machine_post(smcb, ...);
15 }
```

- `PINT_smcb`: State machine control block
- `PINT_client_sm`: Client state machine
- `PINT_smcb_alloc`: Allokiert einen neuen Control Block
  - Dabei wird festgelegt welchen Typ die State Machine hat
- `PINT_sm_frame`: Ein Control Block kann mehrere Frames enthalten
- Die Client State Machine enthält eine Union, die die Informationen für die eigentliche Operation enthält
  - Kann somit für alle Operationen benutzt werden
- Abschließend wird die Client State Machine gepostet, d.h. zur Ausführung gebracht

State Machines

- Operationen werden als State Machines abgearbeitet
  - `.sm`-Dateien enthalten Beschreibung
  - `statecomp`-Werkzeug generiert `.c`-Dateien aus `.sm`-Dateien
  - Eigener Parser und Scanner
- State Machines bestehen aus Zuständen und Übergängen
  - Immer in genau einem Zustand
  - In jedem Zustand wird eine Funktion oder geschachtelte State Machine ausgeführt
  - Rückgabewert bestimmt Übergang
  - Beginn mit erstem Zustand und Ende mit `terminate`- bzw. `return`-Zustand

- Der `terminate`-Zustand beendet die Abarbeitung der State Machine
- Der `return`-Zustand kehrt aus einer geschachtelten State Machine zurück

State Machines... [1]

```
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;
    }
    ...
}
```

- Die Client State Machine für die `create`-Operation startet im `init`-Zustand
  - Darin wird die `create_init`-Funktion ausgeführt
  - Danach wird ein Übergang in den `parent_getattr`-Zustand vollzogen
- Im `parent_getattr`-Zustand wird die geschachtelte State Machine `pvfs2_client_getattr_sm` ausgeführt
  - Tritt kein Fehler auf, erfolgt ein Übergang in den `parent_getattr_inspect`-Zustand
  - Ansonsten wird in den `cleanup`-Zustand übergegangen

State Machines... [1]

```
1  static PINT_sm_action create_init (struct PINT_smcb
       ↪ *smcb, job_status_s *js_p)
2  {
3      struct PINT_client_sm *sm_p = PINT_sm_frame(smcb,
           ↪ PINT_FRAME_CURRENT);
4
5      PINT_SM_GETATTR_STATE_FILL(
6          sm_p->getattr,
7          sm_p->object_ref,
8          PVFS_ATTR_COMMON_ALL | PVFS_ATTR_DIR_HINT |
9              PVFS_ATTR_CAPABILITY |
                   ↪ PVFS_ATTR_DISTDIR_ATTR,
10         PVFS_TYPE_DIRECTORY,
11         0);
12
13     return SM_ACTION_COMPLETE;
14 }
```

- Die Client State Machine enthält eine dedizierte Variable für geschachtelte State Machines vom Typ `getattr`

State Machines... [1]

```
machine pvfs2_client_create_sm
{
    ...
    state cleanup
    {
        run create_cleanup;
        CREATE_RETRY => init;
        default => terminate;
    }
}
```

State Machines... [1]

```
1  static PINT_sm_action create_cleanup (struct PINT_smcb
       ↪ *smcb, job_status_s *js_p)
2  {
3      ...
4      else if (...)
5      {
6          sm_p->u.create.stored_error_code = 0;
7          sm_p->u.create.retry_count++;
8          js_p->error_code = CREATE_RETRY;
9          return SM_ACTION_COMPLETE;
10     }
11     ...
12     PINT_SET_OP_COMPLETE;
13     return SM_ACTION_TERMINATE;
14 }
```

- Über `error_code` wird der Rückgabewert des Zustands gesetzt
  - Dieser Rückgabewert wird zur Bestimmung des Übergangs genutzt
- Der eigentliche Rückgabewert der Funktion gibt an, ob die Ausführung zurückgestellt (`SM_ACTION_DEFERRED`), vollständig ist (`SM_ACTION_COMPLETE`) oder terminiert werden soll (`SM_ACTION_TERMINATE`)

Message Pairs

- Kommunikation über Message Pairs
  - Anfrage durch Client, Antwort durch Server
- Abarbeitung durch spezielle State Machine
  - Name: `pvfs2_msgpairarray_sm`
  - Kümmert sich um Senden und Empfangen
    - Fehlgeschlagene Operationen werden wiederholt
  - Zusätzlich En-/Decoding der Nachrichten
  - Aufruf eines definierbaren Callbacks

State Machines... [1]

```
machine pvfs2_client_create_sm
{ ...
    state create_setup_msgpair
    {
        run create_create_setup_msgpair;
        success => create_xfer_msgpair;
        default => cleanup;
    }
    state create_xfer_msgpair
    {
        jump pvfs2_msgpairarray_sm;
        success => crdirent_setup_msgpair;
        default => cleanup;
    }
    ...
}
```

Message Pairs... [1]

```
1  PINT_msgpair_init(&sm_p->msgarray_op);
2  msg_p = &sm_p->msgarray_op.msgpair;
3  ...
4  PINT_SERVREQ_CREATE_FILL(
5      msg_p->req,
6      sm_p->getattr.attr.capability,
7      *sm_p->cred_p,
8      sm_p->object_ref.fs_id,
9      sm_p->u.create.attr,
10     sm_p->u.create.num_data_files,
11     sm_p->u.create.layout,
12     sm_p->hints);
13 ...
14 msg_p->comp_fn = create_comp_fn;
15 ...
16 PINT_sm_push_frame(smcb, 0, &sm_p->msgarray_op);
```

Message Pairs... [1]

```
1  static PINT_sm_action msgpairarray_post (struct
       ↪ PINT_smcb *smcb, job_status_s *js_p)
2  {
3      ...
4      ret = PINT_encode(&msg_p->req, PINT_ENCODE_REQ,
5                        &msg_p->encoded_req,
                             ↪ msg_p->svr_addr,
6                        msg_p->enc_type);
7      ...
8      ret = job_bmi_send_list(...);
9      ...
10     return SM_ACTION_DEFERRED;
11 }
```

Message Pairs… [1]

- Nachrichten werden für Versand kodiert
  - Bei Empfang wieder dekodiert
- Unterschiedliche Encodings für Anfragen und Antworten
  - Unterschiedliche Felder interessant
  - Beispielsweise für Create-Operation:
    - Anfrage: Dateisystem-ID, Credentials, Datafile-Parameter
    - Antwort: Metafile-Handle und -Attribute
- Unterstützung für mehrere Encoding-Typen
  - Standardmäßig “little endian bytefield encoding”
  - Geschachtelte Makros, die Daten umwandeln und kopieren

Message Pairs... [1]

```
1  endecode_fields_6_struct(PVFS_servreq_create, ...);
2
3  #define endecode_fields_6_struct(name, ...) \
4          endecode_fields_6_generic(name, struct name,
               ↪ ...)
5
6  #define endecode_fields_6_generic(name, sname, ...) \
7          static inline void encode_##name (char **pptr,
               ↪ const sname *x) { \
8              encode_##t1(pptr, &x->x1); \
9              encode_##t2(pptr, &x->x2); \
10             ... \
11         } \
12         static inline void decode_##name (char **pptr,
               ↪ sname *x) { \
13             ... \
14         }
```

Message Pairs... [1]

```
1  static int create_comp_fn (void *v_p, struct
       ↪ PVFS_server_resp *resp_p, int index)
2  {
3      ...
4      PINT_client_sm *sm_p = PINT_sm_frame(smcb,
           ↪ PINT_MSGPAIR_PARENT_SM);
5      ...
6      sm_p->u.create.server_resp.metafile_handle =
           ↪ resp_p->u.create.metafile_handle;
7      sm_p->u.create.server_resp.stuffed =
           ↪ resp_p->u.create.stuffed;
8      ...
9      return 0;
10 }
```

- Dem Callback wird die Antwort des Servers übergeben
  - Dazu wird auf die Eltern-State-Machine zugegriffen und die Daten dorthin kopiert

Zusammenfassung

- Unterstützung für mehrere Schnittstellen
  - MPI-IO, C, POSIX, System Call
- Einfache Installation und Konfiguration
  - Wenige Abhängigkeiten
  - Unproblematisch, da alles im User-Space
- State Machines zur Abarbeitung von Prozessen
  - Unterstützung für geschachtelte State Machines
- Unterstützung für mehrere Verteilungsfunktionen
  - Anpassung an Software-/Hardware-Umgebung
- Unterschiedliche Sicherheitsmechanismen
  - Einfach, schlüssel- und zertifikat-basiert

Quellen I

[1] OrangeFS Development Team. OrangeFS. http://www.orangefs.org/.

[2] OrangeFS Development Team. OrangeFS Documentation. http://docs.orangefs.com/.

[3] OrangeFS Development Team. OrangeFS Wiki. http://www.orangefs.org/trac/orangefs.

[4] Shuangyang Yang, Walter B Ligon III, and Elaine C Quarles. Scalable Distributed Directory Implementation on Orange File System. *7th IEEE International Workshop on Storage Network Architecture and Parallel I/O*, 2011.