Filesystemserver

SDI Gruppe 6

04. Juni 2009

Till Schuberth / Victor van Santen

Überblick

- Einführung
- Entwurfsentscheidungen
- Beschreibung der Server
- Beispielimplementierungen für L4
- Schnittstellenbeschreibung in IDL

Anforderungen an ein Dateisystem

- Speichern von großen Datenmengen
- Daten speichern über Prozessgrenzen und Abschaltvorgänge hinweg
- Strukturiertes Ablegen von Daten
- Paralleler Zugriff auf Daten

Entwurfsentscheidungen

Anforderung an ein L4-Dateisystem:

- Robustheit
- Ausnutzung der L4-IPC-Performance
- Modularer Aufbau
- Keine eingebauten Policys

→ Aufteilung des Servers in zwei Teile: Aufbau eines zustandslosen Servers mit einer Schnittstelle zu anderen Servern, die den Zustand halten

→ Durch den modularen Aufbau ist es möglich, den zustandbehafteten Server später durch einen zustandlosen Server zu ersetzen

Nameserver

- Nameserver löst Namensanfragen auf und gibt zuständige Fileserver zurück (Mountpoints)
- Anfrage wird in rekursive Anfragen, je eine pro Verzeichnis, an den Nameserver überführt
- Im Nameserver werden nur die Mountspoints, die vorher angemeldet wurden, gespeichert.

→ Kleine Datenstruktur im Nameserver , ohne Kenntnisse über die Verzeichnisstruktur

Fileserver

- Fileserver bekommt das „Restverzeichnis“ nach seinem Mountpoint und die Operation vom Client übergeben.
- Er schaut sich dann die tatsächliche Verzeichnis-struktur an und löst diese auf
- Der Fileserver führt die jeweilige Operation aus und liefert die geforderten Daten zurück an den Client

Lockserver

- Der Lockserver nimmt Lockanfragen an und überprüft, ob die Lock verfügbar ist und gibt dann die Lock oder ein „Besetzt“ zurück
- Lockserver lösen Lockfreigaben auf
- Sollte die Operation beim Fileserver ein Lock benötigen (wir bieten auch lockfreie Operationen an) wird dies über den Lockserver abgehandelt

→ Lockzustände nur im Lockserver

→ Kein Zustand im Fileserver

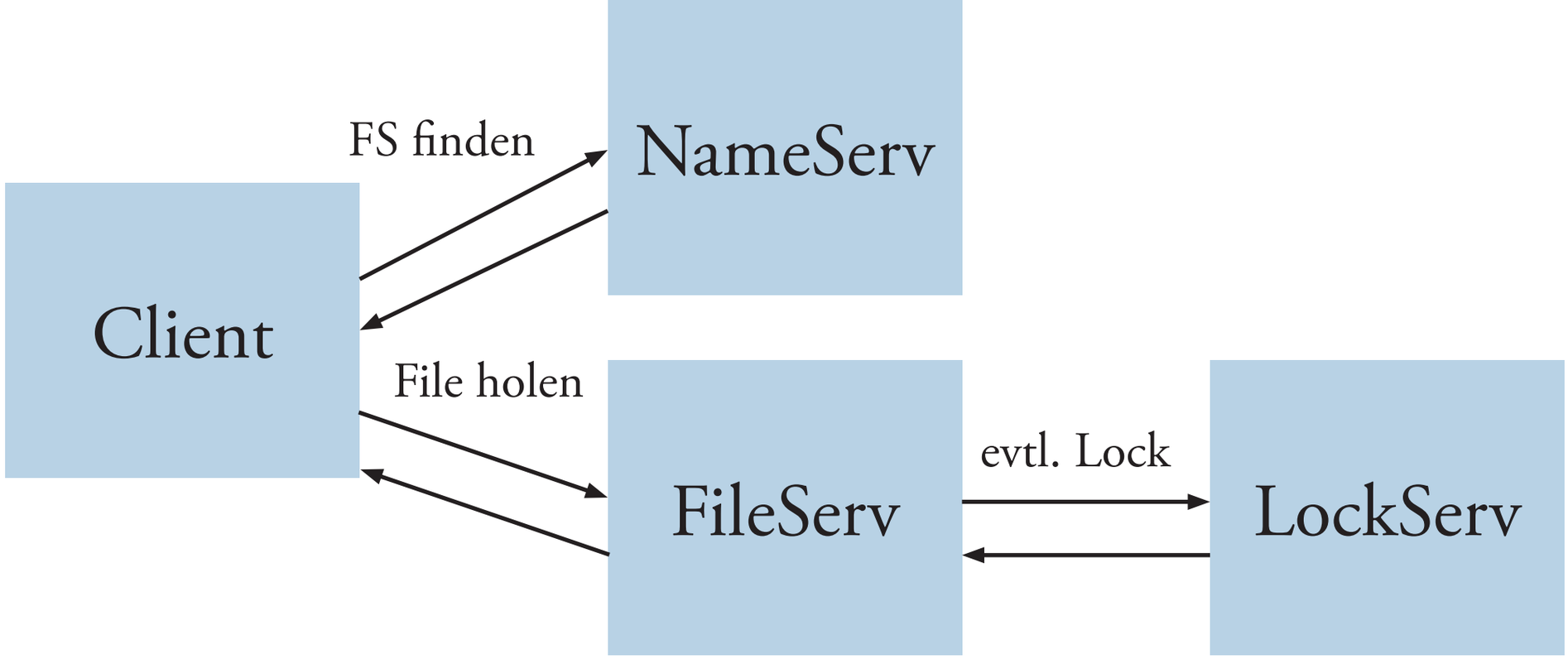
→ Trennung von zustandloser und zustandbehafteter Serverstruktur

Beispiel

- Client führt aus: „*read* /usr/bla/blubb/foobar.txt“
- Nameserver bekommt Anfragen:
  - » /
  - » /usr/
  - » /usr/bla/
  - » /usr/bla/blubb/
- Nameserver liefert zurück:
  - » Rootfileserver
  - » NFS
  - » Null
  - » Null
- Client geht zum NFS, da dieser als tiefster bzw letzter Mountpoint geliefert wurde und stellt die Anfrage:
  - » „*read* /bla/blubb/foobar.txt“

Beispiel (forts.)

- Fileserver durchläuft die Verzeichnisstruktur und versucht im Ordner „Blubb“ die Datei „foobar.txt“ zu lesen. Die gelesenen Bytes werden dann an den Client zurückgeliefert und der Client kann seine Ausführung fortsetzen.



Zusammenspiel der Server

- Client löst mit Hilfe des Nameservers den Namen auf und übergibt Restverzeichnis + Operation an den Fileserver
- Fileserver löst das Verzeichnisnamen auf bis man bei der Datei angelangt ist Internes Filehandle
- Der Fileserver führt die Operation mit dem internen Filehandle aus:
  - Read (inkonsistent) ; Continous Read
  - Write (inkonsistent) ; Continous Write
  - Create ; Delete ; Link
  - SetAttributes ; GetAttributes

Zusammenspiel der Server (forts.)

- Wenn man beim Read/Write das Continous Flag setzt , wird beim Lockserver ein Lock geholt
  - » Falls Lock vergeben Besetzt zurückgeliefert
  - » Falls Lock frei Lock zurückgegeben

Exklusiver Zugriff oder Error zurückgeliefert

Konsistenz erzwungen durch exkl. Zugriff

- Attributoperationen benötigen immer ein Lock!
- Entwurfsentscheidung
  - » Error kann an Client zurückgeliefert werden oder der Fileserver blockiert den Client bis Lock verfügbar

Entwurfsentscheidung im Lockserver

- Bei einem Continous Write wird auf einen exklusiven Zugriff gewartet und dann geschrieben. Es kann hier kein Abbruch erfolgen.
- Bei einem Continous Read wird exklusiv gelesen und im Falle eines Writes wird der Readburst unterbrochen und der Client bekommt eine Exception

→ Bei einem Schreibzugriff warten wir nicht auf alle Lesenden sondern bekommen stets aktuellste Daten

→ Neustart des konsistenten Lesevorgangs möglich

→ Entscheidung liegt aber beim Client in der Implementierung des Exceptionhandlers

Beispielhafte Continous Read Folge

- Read und Write besitzen zwei Flags:
  - » Continous Flag
  - » InSeries Flag

Eine Folge von konsistenten Reads sieht bspw. so aus:

Read [10] , Read [11], Read [11], Read [11], Read [01]

Beginn der Serie	Mitte der Serie	Ende der Serie
C = 1 & IS = 0	C = 1 & IS = 1	C = 0 & IS = 1

- Bei jedem Read in der Serie wird beim Lockserver überprüft, ob es zu einem Write gekommen ist.

IDL-Interface

```
interface fileserver {
        exception access_denied {};
        exception file_not_found {};
        exception file_locked {};
        exception file_exists {};
        exception other_error {};
damit out of memory, not reachable, etc.

struct fs_attr_t {
        unsigned long long size;
        unsigned short owner;
        unsigned short group;
        unsigned short rights;
        unsigned long creation_time;
        unsigned long modification_time;
        unsigned long access_time;
        unsigned short links; (read only)
}
```

IDL-Interface (forts.)

rights: 0x0001: read user
0x0002: write user
0x0004: execute user
0x0008: read group
0x0010: write group
0x0020: execute group
0x0040: read other
0x0080: write other
0x0100: execute group
0x0200: directory (read only)

IDL-Interface (forts.)

```
void read (
    in sequence<char> file,
    in unsigned short flags,
    in unsigned long long offset,
    inout unsigned long size,
    out fpage buffer )
        raises access_denied, file_not_found, file_locked, other_error;
```

flags: 0x01: continuous
0x02: inSeries

- Read kann auch Verzeichnisse öffnen. Die Dateinamen stehen dann mit Null-Bytes getrennt hintereinander.
- Das Lesen hinter einem Dateiende ist daran zu erkennen, dass „size“ auf 0 gesetzt wird.

IDL-Interface (forts.)

```
void write (
        in sequence<char> file,
        in unsigned short flags,
        in unsigned long long offset,
        inout unsigned long size,
        in fpage buffer )
            raises access_denied, file_not_found, file_locked, file_exists, other_error;
```

flags:
- 0x01: continuous
- 0x02: inSeries
- 0x04: create file, error when existing

IDL-Interface (forts.)

```
void create (
        in sequence<char> file,
        in string type )
                raises access_denied, file_not_found, file_exists, other_error;
```

file_not_found: wenn Pfad dort hin nicht existiert

```
void delete ( in sequence<char> file )
                raises access_denied, file_not_found, other_error;

void link (
        in sequence<char> file,
        in sequence<char> dest )
        raises access_denied, file_not_found, file_exists, other_error;
```

- Umbenannt wird durch link und delete

IDL-Interface (forts.)

```
void getAttribute (
        in sequence<char> file,
        out fs_attr_t attr )
        raises access_denied, file_not_found, file_locked, other_error;
void setAttribute (
        in sequence<char> file,
        in fs_attr_t )
        raises access_denied, file_not_found, file_locked, other_error;
```

- Attribute lesen und schreiben benötigt immer eine lock.

}

IDL-Interface (forts.)

```
interface Lockserver {
    void Lock (
        in L4_ThreadId_t thread,
        in unsigned long long Lock,
        out boolean wasLocked );
    void deleteLock (
        in L4_ThreadId_t thread,
        in unsigned long long Lock,
        out boolean wasLocked );
    void isLocked (
        in unsigned long long Lock,
        out boolean isLocked);
```

- Lock: wasLocked wird nur gesetzt, wenn ein Thread die Lock schon hält.
- deleteLock: wasLocked, wenn der angegebene Thread die Lock hat. AnyThread (write access) und AnyLock (thread beendet) sind zugelassen.

```
}
```

Fragen

Fragen?

Danke

Vielen Dank