Naming

Fabian Sperber und Martin Ritter
28.05.2009

Inhalt

- Anforderungen
- Design
- IDL4-Interface
- Root-Service
- Sicherheit
- Client-API
- Beispiel

Anforderungen

- Jedes Objekt im System über stringbasierten Bezeichner eindeutig identifizieren
  - Geräte
  - Dienste
  - Dateien
  - Semaphore/Mutex/Locks/Events
  - Pipes/Mailboxes
  - ...
- (Name-)Services global verfügbar machen
- Mögliche Interface für Objekte abfragen
- Keine Annahme über das Halten eines Status beim Server (in Leaf-Servern mit globalen IDs kann ohne Status ausgekommen werden)

Design: Namensraum

- globaler Namensraum der über mehrere Nameservices verteilt ist
- Jeder Service verwaltet einen Teilbaum des globalen Namensraums und registrieren sich dafür mit createLink()
  - z.B. standardmäßig unter /services/<name>
  - kann sich mehrfach an verschiedenen Stellen mit verschiedenen Einsprungspfaden registrieren
- Jeder Service, der Objekte im globalen Namensraum anbieten will, muss das Nameservice-Interface anbieten (InterfaceID 0)

Design: Objekte

Gewöhnliche Operationen auf Objekten

- Erstellen und Umbenennen
- Öffnen (als Session) oder Identifizieren (global)
- Bearbeiten (Typ-spezifisch)
- Schließen (als Session)
- Löschen

Identifikation über (menschenlesbaren) String (Unicode?). Auflösung von links nach rechts. Jeder Service kann eigene 'Verzeichnis'/Trennzeichen (z.B. '/') Konvention implementieren, jedoch nicht notwendig.

Auflisten aller Objekte in einem Teilbaum kann sehr aufwändig sein und ist nicht unbedingt immer erwünscht =>Service kann extra Aufzählungsobjekt mit eigenem Interface implementieren (siehe Fileserver Beispiel)

Design: Handles

- Beliebige vom Service vorgegebene Bitfolge (→ `L4_Word_t`)
- Handles sind nur für den aufrufenden Client-Thread und Service eindeutig
  - Unterschiedliche Services können gleiche HandleID zurückgeben
  - Ein Service kann mehreren Threads gleiche HandleID zurückgeben, also z.B. global eindeutige Handles verwenden (Filesystem: Inodes)
  - Threads können Handle nicht an andere Threads weitergeben, sondern müssen ihre eigenen erstellen (siehe optionale RPC Befehle)

IDL4-Interface

Auflösen eines Namens und Erstellen eines neuen Objektes im Zielnamensraum:

```
int create(in string name, in Mode mode, out TID threadID, out HANDLE handle, out string targetRoot)
raise(resolveError, doesNotExist, createError, optional securityError, optional encodingError)
```

- `result` - welcher Teil des Pfades schon aufgelöst wurde. Wenn result==strlen(name) ist, wurde der Name komplett aufgelöst. Server versucht so weit wie möglich aufzulösen
- `name` – Name des Objekts
- `mode` – gibt an, ob das Objekt mit diesem Aufruf erstellt werden soll und/oder eine ID dafür angelegt werden soll
  - Erstellt kein Handle: `CreateOnly, CheckExisting`
  - Erstellt Handle: `OpenWithPossibleCreate, OpenWithForceCreate, OpenWithoutCreate`
- `threadID` – ThreadID des Services, der das Objekt bzw. den bisher aufgelösten Pfad besitzt
- `handle` – ObjektID, wenn kompletter Name aufgelöst wurde, sonst ungültig
- `targetRoot` – enthält den Anfangspfad für den (neuen) Nameservice. Dieser muss für weitere iterative Anfragen vor den noch nicht aufgelösten Pfad kopiert werden

Das Interface bevorzugt iterative Auflösung, da dadurch die Server weniger Last und dadurch verfügbarer sind, aber eine Serverimplementation kann sich auch für den rekursiven Ansatz entscheiden (z.B. Virtual Filesystem → Physical Filesystem).

IDL4-Interface

Schließen eines ObjektHandles bzw SessionID mit möglichem Löschen des Objektes

```
void close(in HANDLE handle, in bool delete)
raise(optional securityError)
```

- `handle` – ObjektID eines Objektes
- `delete` – Objekt soll nach `close()` gelöscht werden (bzw Referenzzähler dekrementiert)

Standardmäßig muss nach einem erfolgreichem `open()` das Handle nach Verwendung mit `close()` geschlossen werden. Allerdings kann ein Server auch globale IDs verwenden, die nicht geschlossen werden müssten.

IDL4-Interface

Überprüfen, ob ein Interface für ein Objekt verfügbar ist

```
bool hasInterface(in HANDLE handle, in uint interfaceID)
```

- `result` – Interface verfügbar?
- `handle` – ObjektID eines Objektes
- `interfaceID` – IDL-InterfaceID für ein Objekt (z.B. File Interface)

IDL4-Interface

Erstellen einer Weiterleitung zu einem neuen Nameservice. Nur in aktuellem Nameservice möglich.

```
void createLink(in string name, in string targetRoot)
raise(alreadyExists,notImplemented,optional securityError,optional
encodingError)
```

- `name` – Name des Pfades im angesprochenen Nameservices, der dann folgende Anfragen an aufrufenden Service-Thread weiterleitet
- `targetRoot` – Pfad, der in `create()` als `targetRoot` zurückgegeben wird

Service muss ein spezielles Link-Objekt erstellen, dass auf einen anderen Service weiterleiten kann, wenn diese Funktionalität implementiert wird. Ansonsten muss mit einer notImplemented-Exception beendet werden.

Das Link-Objekt ist mit einem Handle erreichbar, daher ist eine deleteLink() oder ähnliche Funktion nicht nötig, da über close jedes Objekt, also auch Weiterleitungen gelöscht werden können.

Rückgabe der ThreadID eines Link-Objektes

```
L4_ThreadId_t getLinkThread(in handle) raise(noValidLink)
```

Nur verwendbar auf Link-Objekt-Handle, sonst Exception. Kann verwendet werden, um die ThreadID eines Services herauszufinden, ohne ein Objekt in dessen Namensraums aufzulösen.

IDL4-Interface (optional)

Mögliche zusätzliche Funktionen

```
void checkLink(in string name)
```

benachrichtige Nameservice, dass die Weiterleitung nicht mehr verfügbar ist. Daraufhin überprüft der Server selbst noch einmal, ob der entsprechende Service erreichbar ist und löscht gegebenenfalls den Eintrag

```
void createFromHandle(in HANDLE handle, in TID threadID, out HANDLE newHandle)
raise(notExists,optional securityError)
```

erstellt neues Handle für aufrufenden Thread aus einem Handle eines anderen Threads

Root-Service

Finden des ersten Nameservice-Threads

- Statische ThreadID (unflexibel, aber einfache Implementierung)
- Zusätzlicher Parameter für main() Methode (Elf-Loader-Service ö.ä. muss dafür modifiziert werden)
- Eintrag in KIP (wäre schön, aber wohl nicht möglich ;))
- Gemappte Page (feste Addresse, Platzverschwendung 4KB vs 1 Word)
- Userlibrary → „Dynamic Binder Magic“

Sicherheit & Authentifizierung

- Wird durch eigenen Service implementiert
- Jeder Service, der sich um Sicherheit kümmert, sollte sich bei diesem Service melden und Client-seitige Anfragen verifizieren
- Kein besonderes Protokoll für Nameservice nötig.
- Bestehende Funktionen können höchstens fehlschlagen, daher `raise(securityException)`

Client-API

Da die Namensauflösung iterativ und die direkte Verwendung der RPC Funktionen umständlich sein kann, sollte eine Userlibrary angeboten werden, die die meistverwendeten Operationen anbietet.

```
bool Create(string name)

bool Rename(sting name, sting newname)

bool Open(string name, word *handle,L4_ThreadId_t
*threadID)

bool Close(word handle)

bool Delete(string name)

bool Exists(string name)
```

Convenience-Funktion Open

```
bool open(string name, HANDLE &handle, TID &serverTID)
{
    serverTID = RootNameServerThreadID;
    string nextRoot = '';
    int resolved = 0;
    while (true)
    {
        name = nextRoot + name[resolved:strlen(name)];
        resolved = serverTID->create(name, OpenWithoutCreate,
serverTID, handle, nextRoot);
        // check exceptions... → return false
        If (resolved==strlen(name)) return true;
    }
}
```

Beispiel Fileserver

```
if (open('vfs/usr/bin/gcc', handle, server))
{
    if (server->hasInterface(handle, FILE_INTERFACE_ID))
    {  // for example
        file->useAccess(handle, READ | WRITE); // get read+write
access rights for this session
        // do work: read, write, …
    }
    if (server->hasInterface(handle, DIRECTORY_INTERFACE_ID))
    {
        // do work: list directory entries,...
        directory->getCount();
        entryname = directory->getName(index);
    }
    server->close(handle, false);
}
```