Apertos

ein objektorientiertes Betriebssystem

Markus Hanke

26.11.1997

Hauptseminar WS 97/98:
Aktuelle Trends in der Betriebssystem-Forschung

1. Überblick

Apertos ist ein objektorientiertes (Forschungs-) Betriebssystem, das sich die neuen Ansätze zur Reflexion in objektorientierten Programmiersprachen zu nutze machen will. Es wird derzeit am Department of Computer Science, Keio University, entwickelt. Das System hat den Anspruch, durch hohe Flexibilität den Aufbau und die Verwaltung von engverzahnten großen Rechnernetzen zu ermöglichen (bzw, gegenüber traditionellen Systemen zu verbessern). Dieses Papier will einen kurzen Überblick über dieses System geben. Es geht dabei insbesondere auf die Strukturierung des Systems in Objekte, Metaobjekte und Metaräume ein und wie diese Konstrukte Reflexion ermöglichen. Es geht darauf ein, wie die Anwendunge dieser Reflexion die Flexibilität des Systems erzeugt, und, da Reflexion im Grundmodell viel Rechenzeit kostet, wie man gerade mit der Reflexion Rechenleistung sparen, und die Effizienz erhöhen kann.

Das 2. Kapitel führt kurz in die Terminologie von Metaberechnungen und Reflexion ein. Im 3. Kapitel wird ein Objekt, wie es in Apertos verwendet wird definiert. Das 4. Kapitel verbindet nund diese beiden Punkte, und baut damit die Konstruktion auf, die das Grundgerüst von Apertos ausmacht. Im 5. Kapitel wird dieses Gerüst dann konkretisiert, und dargestellt, wie es in aktuellen Implementationen von Apertos zur Anwendung kommt. Welche Programmiersprachen zur Implementation von Apertos verwendet wurden, bzw verwendet werden wird im 6. Kapitel dargelegt. Auf das eigentliche Ziel von Apertos, nämlich Flexibiltät bei ausreichender Effizienz, kommt das 7. Kapitel zurück.

2. Metaberechnungen und reflexive Berechnungen.

Man unterscheidet Berechnungen auf der Objektebene und Berechnungen auf der Metaebene, wobei in diesem Papier die Objektebene als Basisebene bezeichnet wird, da die Bezeichnung „Objekt“ bereits anderweitig bemüht wird. Berechnungen der Basisebene beziehen sich auf den eigentlichen Gegenstand der Berechnungen. Berechnungen auf der Metaebene sind solche, die sich auf die Basisebene beziehen. So könnte man zum Beispiel ein Programm zum Faktorisieren großer Zahlen in zwei Teile untergliedern. Der erste beschäftigt sich mit den verschiedenen Problemen der Faktorisierung. Der zweite beschäftigt sich damit, wie man diese Berechnungen parallelisieren kann, bzw. auf verschiedene Knoten verteilen kann, oder welche Prozessoren für welche Teilbereiche am besten geeignet sind, und verteilt die Berechnungen dementsprechend. Den ersten Teil würde man in diesem Zusammenhang als Basisteil, den zweiten als Metateil bezeichnen.

Reflexiv nennt man die Metaberechnungen, wenn sie mit den selben Methoden geleistet werden, wie die Berechnungen auf der Basisebene. So leistet zum Beispiel ein Interpreter zwar Metaberechnungen (bezogen auf die Basisebene des zu interpretierenden Programms), aber keine reflexiven Berechnungen, da die Metaberechnungen auf einer komplett anderen Ebene angesiedelt sind, als das Programm selbst. Das Faktorisierungsprogramm dagegen ist reflexiv angelegt. Oder anders formuliert: Der Interpreter macht Berechnungen über ein anderes Programm, während das Faktorisierungsprogramm Berechnungen über sich selbst anstellt.

3. Das Objekt in Apertos

Bei Apertos handelt es sich um ein objektorientiertes Betriebssystem. Die kleinste, und wichtigste Einheit, mit der das System arbeitet (scheduled, benennt, verschiebt, erzeugt, löscht, ...) ist ein (Apertos-) Objekt. Ein Objekt in Apertos ist durch folgende Eigenschaften gekennzeichnet: Es hat eine Klasse, und somit eine Schnittstelle, die anderen Objekten zumindest soweit bekannt ist, wie sie eine der Superklassen kennen. Weiterhin hat es genau einen Aktivitätsträger, womit bereits festgelegt ist, dass eine explizite Synchronisation für Apertosprogramme komplett entfällt. Das Objekt ist in Apertos auch die kleinste Einheit, die Schutz durch das Betriebssystem erhält (z.B. einen eigenen Adressraum erhalten kann). Außerdem besitzt jedes Objekt eine Identität, die Systemweit eindeutig vergeben wird, und sich während der gesamten Lebensdauer des Objekts nicht verändert.

Um zu kommunizieren tauschen Objekte untereinander Botschaften aus. Davon gibt es prinzipiell zwei Typen. Synchrone Botschaften, bei denen das sendende Objekt wartet, bis das Empfangende den Rückgabewert berechnet und zurückgeschickt hat, sowie asynchrone Botschaften, bei denen der Sender sofort weiterrechnet, und ein eventuell vorhandener Rückgabewert, nachdem er berechnet wurde in einer weiteren Botschaft zurückgesendet wird.

4. Metaräume

In Apertos hat jedes Objekt (Systemobjekte und auch Applikationsobjekte) einen Metaraum. Dieser setzt sich aus einer Menge von Metaobjekten zusammen. Der Metaraum definiert die Ablaufumgebung für ein Objekt. Er bestimmt gewisse Teile der Semantik der Objekte, wie zum Beispiel die Schedulingstrategie, oder die Art und Weise, in der Objekte gespeichert werden (persistent oder nicht). Die Liste der Eigenschaften, die ein Metaraum einem Objekt verschaffen kann, ist sicherlich offen. Derzeit umfasst sie hauptsächlich folgende Gebiete: Scheduling (z.B. verschiedene Strategien, oder Realzeiteigenschaften), Speicherung (z.B. das Medium, auf dem das Objekt gespeichert ist (Hauptspeicher oder Platte)) sowie die Semantik von Methodenaufrufen. Der Metaraum heißt deshalb Raum, da er sich aus mehreren (Meta-) Objekten zusammensetzt. Ein Programmierer, der ein neues Objekt erstellt, hat also die Freiheit, sich einen geeigneten Metaraum auszuwählen. Er kann zum Beispiel beliebige vorhandene Abstraktionen benutzen, oder sie umgehen (soweit das die Sicherheit z uläßt).

Allerdings ist die Beziehung zwischen den Objekten und ihrem Metaraum eine beiderseitige. So weiß nicht nur jedes Objekt über seinen Metaraum Bescheid, welche Eigenschaften dieser hat, welche Dienste er anbietet, und welche Schnittstellen dafür zur Verfügung stehen, sondern auch der Metaraum kennt viele, für das System an dieser Stelle, bzw. in diesem Metaraum, wichtige Eigenschaften der Objekte. Das heißt dass die Menge dieser Eigenschaften nicht fest ist, wie dies in herkömmlichen Betriebssystemen (UNIX, Mach, ...) der Fall ist. Dort sind die Möglichkeiten, das Verhalten der Anwendung zu beeinflussen, bzw. Informationen über die Anwendung herauszubekommen, sehr begrenzt. Einige der Informationen, die der Metaraum über das Objekt bekommt, ergeben sich schon aus der Wahl des Metaraums selbst. Andere sind in der Klasse des Objekts enthalten, die dem Metaraum zugänglich ist. Eine dritte Informati-

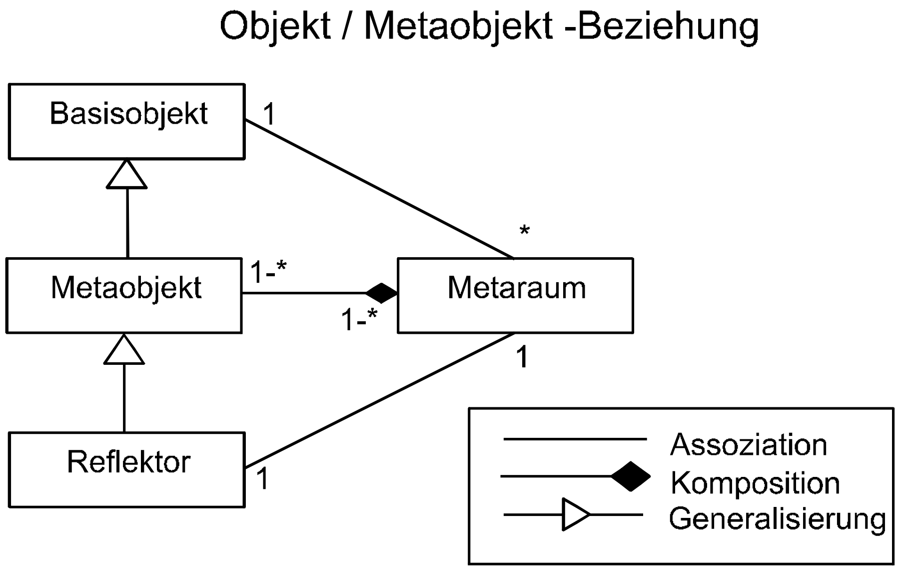

Abbildung 1 : Die Beziehungen der Entitäten in Apertos

onsquelle bietet die Statistik der Botschaftsübermittlung zwischen Objekten, da alle Botschaften zwischen den Basisobjekte über den Metaraum abgewickelt werden. Diese beidseitige Beziehung zwischen Objekten ist wohl einer der wesentlichen Unterschiede zu vielen anderen Betriebssystemen. Dies ist auch der Grund, weshalb die Metaobjekte so genannt werden. Aufgrund der Informationen, die der Metaraum, bzw. die ihn konstituierenden Metaobjekte haben, sind sie in der Lage darauf aufbauend Metaberechnungen über die Basisobjekte zu machen.

Eine bedeutende Folge dieses Umstands ist es, dass Objekte ihren Metaraum wechseln können. Wenn ein Objekt seinen Metaraum wechseln will, so meldet es diesen Wunsch bei seinem Metaraum an, der dann zusammen mit dem Zielmetaraum die Kompatibilität des Objekts mit dem Ziel überprüft, und, im positiven Fall, das Objekt in den Zielraum verschiebt, wo es dann seine Ausführung fortsetzen kann. Der Wunsch, seinen Metaraum zu wechseln, kann zum Beispiel dadurch entstehen, dass das Objekt entweder gewisse Semantiken seiner Ausführung ändern will (z.B. Scheduling-Strategie, Seitenersetzungsstrategie), oder in einen anderen Adressraum oder Knoten wechseln will. Diesen Vorgang bezeichnet man als Objektmigration.

Jedem Metaraum ist ein Reflektor zugeordnet, der den Metaraum konstituiert. Er hält die Liste der Referenzen zu den Metaobjekten des Raums. Er stellt somit einen Einsprungpunkt dar. Über ihn nehmen Objekte zu ihrem Metaraum, und den darin enthaltenen Objekten Kontakt auf. (Da zwischen Metaräumen und Reflektoren eine 1-1:1-1 Beziehung besteht, bezeichnen wir ab jetzt Metaräume immer nach ihren Reflektoren. Die Namen von Reflektoren beginnen mit einem kleinen m.)

An der 1-n:1-n - Beziehung zwischen Metaobjekt und Metaraum erkennt man, dass durchaus ein Metaobjekt zu mehreren Metaräumen gehören kann.

5. Die Metahierarchie

Da Metaobjekte auch Objekte sind, und Objekte in Apertos prinzipiell einen Metaraum besitzen, führt dies zu einer Hierarchie von Objekten und ihren Metaräumen bzw. Metaobjekten. Im Detail: Jedes Objekt hat einen Metaraum, welcher aus mehreren Metaobjekten besteht. Jedes dieser Metaobjekte hat wiederum einen Metaraum (also eigentlich einen Meta-Metaraum). Zu beachten ist hierbei, dass nicht etwa der ganze Metaraum einen Metaraum hat, sondern jedes einzelne Metaobjekt hat seinen Metaraum.

Dazu als kurzes Beispiel, die Speicherverwaltung in Apertos (Abbildung 2):

Der Speicher von Anwendungsobjekten wird von mehreren Segmentobjekten gehalten. Eine Anwendung sucht sich ihre Segmentobjekte danach aus, welche speziellen Eigenschaften sie anbieten, zum Beispiel welche Seitenersetzungsstrategie es verfolgt. Der komplette (virtuelle) Speicher, wird auf die Segmentobjekte, als Einheiten zur Seitenverwaltung, aufgeteilt. Auch die Segmentobjekte haben ihre Metaobjekte. Dies sind in diesem Fall die Page-Objekte. Sie halten den Speicher, der von den Segmentobjekten benötigt wird, sowie bestimmte Metainformationen über die Segmentobjekte. Sie sind so programmiert, dass sie nicht in einen sekundären Speicher verschoben werden.

Da die Metahierarchie prinzipiell unendlich ist, dies aber für ein Betriebssystem nicht unbedingt praktisch ist, wird sie in Apertos auf spezielle Weise abgeschlossen: Die beiden obersten Metaräume (mCore und mZero) dienen sich gegenseitig als Metaraum. Alle anderen Reflektoren haben mZero (entweder direkt oder indirekt) als Metaraum.

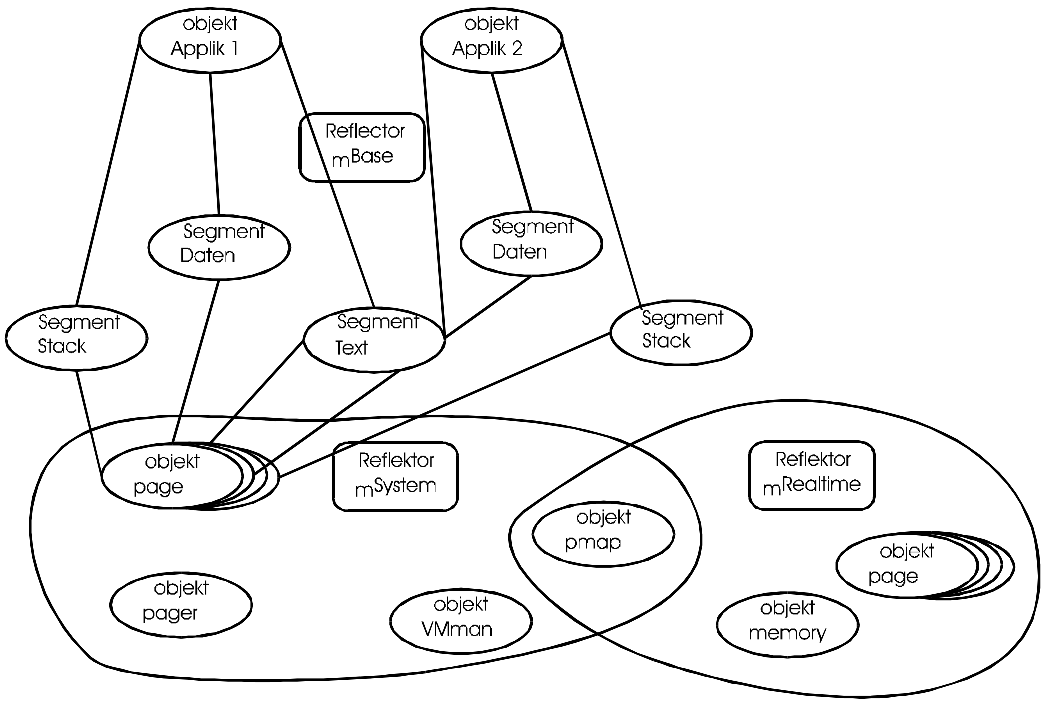

Abbildung 2: Beispiel Speicherverwaltung

Dazu zählen unter anderem: mDrive (Metaraum für Gerätetreiber), mSystem (Metaraum für Systemobjekte), mStorage, mPersistent und mRealtime.

Da Reflektoren Objekte sind, die aus Klassen erzeugt werden, und diese Klassen vererbbar sind, gibt es für diese auch eine Vererbungshierarchie. Einige der Reflektorklassen (die wohl deshalb genauso wie die Reflektoren heißen, weil von diesen Klassen jeweils nur ein Objekt instantiiert wird) und ihr Platz in der Vererbungshierarchie stellt Abbildung 3 dar.

Hierbei ist **mCommon** nur eine abstrakte Basisklasse, die nicht instantiiert werden kann. Er stellt unter anderem Methoden zur Objektmigration (siehe Kapitel 3), wie zum Beispiel zum testen der Kompatibilität zwischen dem Quell- und dem Zielraum. **mBase** ist ein Reflektor, der bereits zum engeren Kernbereich von Apertos gehört. In Hardwarearchitekturen, die zwischen Benutzer- und Systemspeicher unterscheiden, ist mBase für den Benutzerspeicher verantwortlich. Er bietet eine Warteschlangen-Schedulingstrategie innerhalb einer Prioritätsklasse an. Das heißt, dass Objekte nur von höherprioritären Objekten unterbrochen werden können. **mZero** ist der Reflektor des Metaraums, der alle anderen Reflektoren als Basisobjekte hat (beachte: da Reflektoren eine Spezialisierung von Objekten sind, haben sie, wie gewöhnliche Objekte auch, einen Metaraum). Alle Botschaften von zwischen Objekten, die nicht im selben Metaraum sind, laufen über mZero. Als letztes soll **mCore** noch besonders erwähnt werden. mCore hat viele Aufgaben, die auch herkömmliche Kerne erfüllen. Unter anderem muss er die Schedules der verschiedenen Metaräume, implementieren. Das heißt er muss sie, gemäß einer ihm eigenen Strategie, mischen.

Der Kern von Apertos wird MetaCore genannt. Er stellt nur fünf Funktionen zur Verfügung, und ist damit wirklich ein Nanokern. Die fünf Funktionen sind die folgenden:

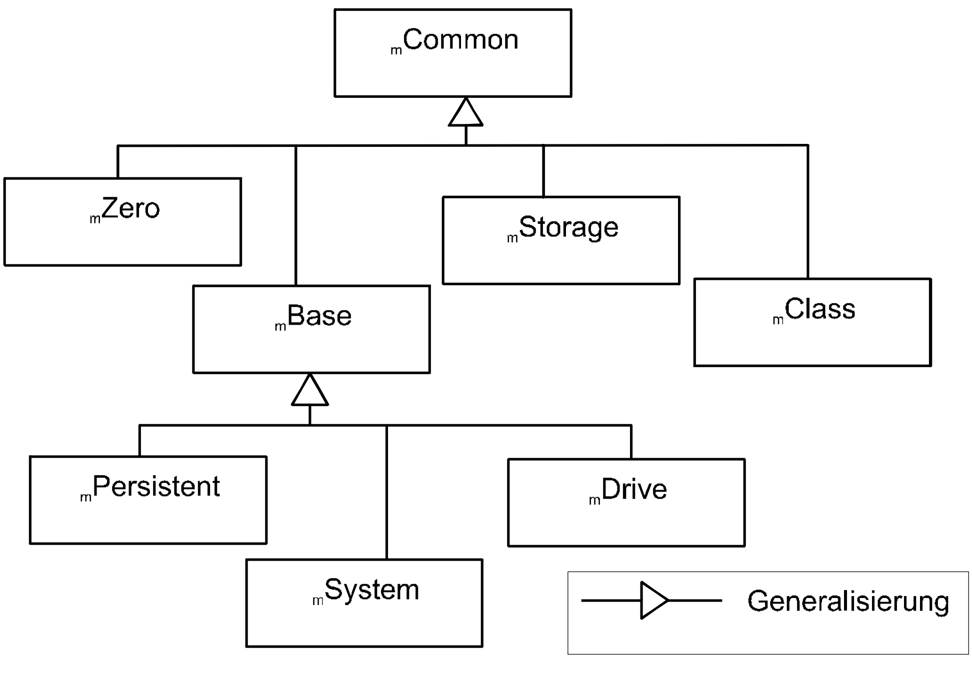

Abbildung 3: Vererbungshierarchie der Reflektorklassen

- M(eta) stoppt die Ausführung des aktiven Objekts und gibt sie an das Metaobjekt weiter.
- R(eflekt) beendet die Berechnung des Metaobjektes und setzt die Berechnung des Basisobjektes fort.
- CActive gibt eine Referenz auf das gerade aktive O bjekt zurück.
- CBind verbindet einen Hardwareinterrupt mit einem Objekt.
- CUnBind löst diese Verbindung wieder.

Dies sind also nur die notwendigsten Funktionen (die überdies in ihre Ausführungsdauer vorhersagbar, und damit echtzeitfähig sind), die benötigt werden, um das Konzept der Objekt / Metaobjekt-Trennung zu ermöglichen.

6. Verwendete Programmiersprachen

Die ersten Implementierungen der Objekte erfolgte mit C++. Dies hatte allerdings einige Nachteile. Das Hauptproblem war, dass die C++-Objekte nicht mit den Apertos-Objekten korrespondierten. Man hat also zum Beispiel Methodenaufrufe von Apertosobjekten anders getätigt als Methodenaufrufe von C++-Objekte. Die Methodenaufrufe von Apertosobjekte benötigten jeweils einen Stub, der vom Programmierer in den C++-Text eingefügt werden musste. Eine Erleichterung erbrachte die Einführung von MC++, eine eigens für Apertos abgewandelte Form von C++, bei der ein weiteres Schlüsselwort eingeführt wurde, das man an Methodenaufrufe von aktiven (Apertos-) Objekten anhängte. Ein Präcompiler erzeugte dann den nötigen Stub. Allerdings blieb damit die Unterscheidung zwischen C++- und Apertosobjekten bestehen. Dieser Unterschied macht sich hauptsächlich dadurch bemerkbar, dass passive C++ Objekte nicht bei Methodenaufrufen an andere Apertosobjekte mitgegeben werden, da ihr Code in den Code von dem sendenden Apertosobjekte compiliert ist, und somit dem Empfangsobjekt nicht zur Verfügung steht. Um dieses Problem zu lösen, ist also eine stärkere Anpassung der verwendeten Programmiersprache an Apertos nötig.

Diese wurde geleistet mit der Entwicklung der Programmiersprache Cognac. Als Grundlage von Cognac wurde Smalltalk80 verwendet. Der Grund dafür ist, dass der Aufrufsyntax von Smalltalk80 sehr gut zu Apertos passt. Es wurden jedoch einige Modifikationen durchgeführt:

- Optionale Typdeklarationen für Variablen, Methodenparameter und Rückgabewerte. Allerdings wurde dazu ein Untertypensystem benutzt, um den Programmierer nicht unnötig einzuschränken. Die eingeführten Typen sind hauptsächlich dazu da, die Effizienz zu steigern.
- Ein Ausdruck zum asynchronen Methodenaufruf (der schließlich eine wesentliche Eigenschaft von Apertosobjekten ist) sowie
- eine dazugehörige asynchrone Antwortbotschaft.

Um das eigentliche Problem, des Auseinanderklaffens von Programmiersprachobjekten und Apertosobjekten, zu beseitigen wurde folgender Ansatz gewählt: Man überläßt die Entscheidung, ob ein Objekt nun ein reines Programmiersprachobjekt, oder ein Apertosobjekt wird, dem Compiler, der je nach Verwendung eine der zwei folgenden Möglichkeiten ergreift: 1. Das Objekt wird niemals als Apertosobjekt eingesetzt, und wird somit statisch als Programmiersprachobjekt eingelinkt. Seine Methodenaufrufe sind gewöhnliche Prozeduraufrufe. 2. Das Objekt wird auch als Apertosobjekt eingesetzt. In diesem Fall kann sich das Objekt, je nach aktuellem Einsatz, unterschiedlich verhalten. Um in diesem Bereich größtmögliche Flexibilität zu erreichen, wurde ein Zwischencode eingeführt. Das heißt der Compiler erzeugt nicht direkt aus dem Quelltext den Maschinencode, sondern generiert zuvor einen Zwischencode, der zusammen mit dem Maschinencode im Objekt gespeichert wird. Mutiert nun ein Objekt zum Apertosobjekt, so kann der Compiler zum Beispiel den Zwischencode neu compilieren, so dass das Objekt den neuen Anforderungen gerecht werden kann. Die Einführung dieses Zwischencodes hat unter anderem den selben angenehmen Effekt wie bei Java auch: Portierbarkeit. Solchermaßen compilierte Objekte können sehr leicht auf Knoten mit andere Hardwarearchitektur migrieren.

Der Compiler hat außerdem den großen Vorteil, dass er semantische Informationen, die bereits zur Compilerzeit feststehen, statisch in das Objekt hineinbinden kann. Das heißt dass, bei entsprechend statischen Objekten, große Teile des Metaraums in das Objekt hineincompiliert werden. Dies hat den enormen Vorteil, dass ein erheblicher Kommunikationsaufwand eingespart wird, und somit die Effizienz des Systems erheblich gesteigert wird.

7. Ziel bzw. Zweck dieser Architektur

Flexibilität

Die Konstrukteure dieser Architektur erhoffen sich unter anderem eine wesentlich erhöhte Flexibilität des Systems. Diese entsteht an mehreren Stellen des Systems. Zu erst einmal in der Objektstruktur des Systems, die es möglich macht an vielen Stellen des Systems eigene, durch Vererbung leicht veränderte Objekte zu plazieren. Ein weiterer Faktor sind die Metaräume. Dadurch ist es möglich, ohne viel Umstände, aus bereits vorhandenen und/oder neuen Objekten, neue Metaräume zu konstruieren, die sich ihren Basisobjekten mit veränderten Eigenschaften präsentieren, um veränderte Anforderungen zu erfüllen. Einen wichtigen Beitrag dazu liefert die Möglichkeit, dass Objekte ihren Metaraum wechseln können. Damit können auch bereits laufende Objekte ihren Metaraum wechseln.

Hierzu ein Beispiel: Ein Benutzer arbeitet an einer Arbeitsstation, die mit dem Rechnernetz des Büros verbunden ist. Er hat eine ganze Anzahl von Anwendungen gestartet, die möglicherweise eng miteinander verknüpft sind. Nun möchte dieser Benutzer, das Haus verlassen und eine Bahnfahrt antreten, während er auf seinem Laptop weiterarbeiten will. In den bereits erwähnten herkömmlichen Betriebssystemen, muss nun der Benutzer zuerst alle Anwendungen beenden. Danach muss er entscheiden welche Daten - die üblicherweise in Form von Dateien gespeichert sind - er benötigt, um weiterarbeiten zu können. Danach kopiert er diese auf seinen Laptop. Im Zug startet er nun wieder alle Anwendungen, mit denen er auf dem Laptop weiterarbeiten kann (einige fallen sicherlich aufgrund der gekappten Netzverbindung heraus), und arbeitet dann weiter. Dieser komplette Arbeitsgang, der leicht 15 Minuten dauern kann, kann im Prinzip vom System übernommen werden. Denn aufgrund ausreichender Metainformationen über die laufenden Anwendungen kann das System selber entscheiden, welche Daten - die hier in Form von persistenten Objekten gespeichert sind - benötigt werden, und welche Anwendungen mit dem Zielrechner (bzw. dort vorhandenen Metaräumen) kompatibel sind (die, die das Netzt benötigen z.B. nicht). Diese können dann in den neuen Rechner migrieren, und können sofort weiterlaufen, ohne gestoppt und neu gestartet werden zu müssen.

Effizienz

Die gleichen Methoden, die im letzten Absatz besprochen wurden, um die Flexibilität des Systems zu ermöglichen, können auch dazu benutzt werden, die Effizienz des Systems zu erhöhen. Dazu gibt es viele Gelegenheiten. Einige davon schafft die Objektmigration. Diese kann zum Beispiel dazu benutzt werden, Objekte, die sehr viele Botschaften untereinander austauschen, an zueinander nahe gelegene Orte zu transportieren. Das heißt, entweder auf den gleichen Knoten, oder sogar, soweit nicht andere Interessen wie zum Beispiel Sicherheit dagegensprechen, in den gleichen Adressraum. In diesem Fall besteht sogar die Möglichkeit den kompletten Kommunikationsoverhead zu entfernen, und aus der Botschaftsübermittlung einen einfachen Prozeduraufruf zu machen. Weiter Möglichkeit mit Hilfe der Objektmigration die Effizienz zu erhöhen, bestehen darin, durch verschieben der Objekte an andere Knoten eine Lastbalancierung zu erzielen. Auch beim Scheduling ergeben sich Gelgenheiten. Stellt man fest, dass einige Objekte kaum parallel laufen, so kann man diese Gruppieren und in einem gemeinsamen Kontext ablaufen lassen, so dass häufige Kontextwechsel vermieden werden können. Andererseits kann man Objekte, bei denen man feststellt, dass sie parallel agieren können, in Metaräume bringen, die mehrere Kontexte haben, und so echte Parallelität ermöglicht wird. Ein weiteres Beispiel soll noch angebracht werden: Werden von einem Objekt zu einem anderen viele asynchrone Botschaften versendet, kann man diese Stapeln, und in einem einzigen Kommunikationsvorgang übermitteln.

Man sieht, dass die Metainformation, die man über Objekte hat, zusammen mit der flexiblen Gestaltung von Metaräumen, sowie der Möglichkeiten der Objektmigration ein großes Potential zu Effizienzsteigerung ergeben.