Design und Implementierung eines verteilten Objektmanagers für einen wartungsfreundlichen Linux Workstation Cluster

Martin Waitz

Design und Implementierung eines verteilten Objektmanagers für einen wartungsfreundlichen Linux Workstation Cluster

Studienarbeit im Fach Informatik

vorgelegt von

Martin Waitz,

geboren am 25. April 1976 in Coburg

Angefertigt am:

Lehrstuhl für Informatik III
Institut für Mathematische Maschinen und Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dr.-Ing. Volkmar Sieh**
Beginn der Arbeit:	1. Februar. 2000
Abgabe der Arbeit:	15. Dezember 2000

Ich versichere, dass ich meine Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.

Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 15. Dezember 2000,

Inhaltsverzeichnis

1 Einleitung

In vielen Firmen und Instituten stehen eine große Anzahl Arbeitsplatzrechner herum, die während der täglichen Routinearbeit und vor allem nachts nur schlecht ausgelastet sind.

Auf der anderen Seite werden aber häufig große und teure Maschinen gekauft, von denen die prozessorintensiven Arbeiten erledigen werden müssen und dabei völlig überlastet sind.

Wenn man bedenkt, dass bei modernen Hochleistungsrechnern dazu übergegangen wird, diese aus vielen Standardrechnern zusammenzubauen, ist man versucht, auch die schon vorhandenen Rechner zu einem Cluster zusammenschließen, und die zu erledigenden Aufgaben auf diesen aufzuteilen.

Moderne Betriebssysteme können schon jetzt sehr gut von der vorhandenen Hardware abstrahieren und bieten den Anwendungsprogrammen eine standardisierte Sicht der vorhandenen Ressourcen.

Nur bei Rechnerclustern ist dies noch nicht sehr weit verbreitet. Es ist im Allgemeinen sehr schwierig, die Leistung eines Verteilten Systems wirklich auszunutzen. Die Programmierung dafür ist sehr aufwendig, die komplette Kommunikation und Synchronisation zwischen den einzelnen Rechnern (hier Knoten genannt) muss selber erledigt werden.

Das Ziel eines Cluster-Betriebssystems sollte es sein, alle auf den Knoten vorhandenen Ressourcen allen Applikationen auf die gleiche Art und Weise bereit zu stellen. Die Anwendungen sollen die Knotengrenzen im Idealfall überhaupt nicht mehr sehen können, für diese erscheint der Cluster dann wie eine große Multiprozessormaschine.

Dazu müsste es im Betriebssystem eine einheitliche Schnittstelle geben, mit der auf die einzelnen Ressourcen (zum Beispiel Speicherseiten, Plattenblöcke, etc.) von allen Knoten aus zugegriffen werden kann.

Gegenstand dieser Studienarbeit ist es, auf die hierfür benötigten Komponenten einzugehen. Dazu gehören ein Kommunikationsmodul, ein Objektmanager zum Verwalten der einzelnen Ressourcen sowie ein Knotenmanager, mit dem die Knoten auch komfortabel in das System eingebunden werden können.

2 Kommunikation

Um Ressourcen mit anderen Knoten teilen zu können, müssen die einzelnen Knoten zunächst einmal zur Verständigung untereinander fähig sein.

Die Kommunikation in einem verteilten System nimmt eine wichtige Rolle ein, da doch viele Zugriffe auf andere Ressourcen nur über Kommunikation zu anderen Knoten möglich sind.

Andere Knoten müssen informiert werden können, wenn Ereignisse eintreten, die für sie von Bedeutung sind. Auch muss eine Möglichkeit gegeben werden, um Ressourcen auf einen anderen Knoten transferieren zu können. Die Verwaltung dieser Ressourcen erledigt der weiter unten beschriebene Objektmanager. Diesem muss durch das Kommunikationsmodul eine Möglichkeit gegeben werden, auf andere Knoten in dem System zuzugreifen.

2.1 Kommunikationsform

Es gibt verschiedene Arten, wie die Kommunikation zwischen den einzelnen Knoten gestalten werden kann.

Ereignisbasiert: Einem Knoten wird die Nachricht geschickt, dass ein bestimmtes Ereignis eingetreten ist. Diese Nachricht enthält alle Daten, die benötigt werden, um das Ereignis genau zu beschreiben.

Der Knoten kann dann – abhängig von seinem derzeitigen Zustand – darauf reagieren. Dies beinhaltet eventuell das Senden von weiteren Nachrichten und/oder einem Zustandsübergang des Knotens.

Mit Hilfe solcher Ereignisse kann das Programm als endlicher Automat modelliert werden.

Fernaufrufe: Dies ist vergleichbar mit einem Funktionsaufruf, nur dass der Prozess die Funktion auf dem entfernten Knoten ausführt. Zu diesem Zweck wird der Aktivitätsträger auf dem anderen Knoten transferiert und dort die gewünschte Funktion aufgerufen. Danach kehrt der Aktivitätsträger wieder zurück, und das Ergebnis der Funktion ist auf dem aufrufenden Knoten verfügbar.

Auf diese Weise ist es möglich, die Kommunikation zu verbergen, und weiter „traditionell prozedural“ zu programmieren.

In dem hier vorgestellten System wird ein Zwischenweg dieser beiden Ansätze beschritten.

Wenn auch Prozesse als Objekt repräsentiert werden, ist es für diese möglich, einen Fernaufruf dadurch zu implementieren, dass sie sich selber (also den Aktivitätsträger) auf den gewünschten Knoten verschieben, dort die Funktion aufrufen und sich anschließend wieder zurück auf den Ursprungs-Knoten begeben.

Dies erlaubt eine sehr saubere und elegante Möglichkeit für das Betriebssystem, Arbeiten auf verschiedenen Knoten auszuführen. Nur durch diese Trennung der Prozesse von einzelnen Knoten ist es möglich, den Anwendungen einen transparenten Zugriff auf alle Knoten zu geben.

Es bedeutet aber auch, dass das Verwalten der einzelnen Objekte nicht mehr in einem eigenen Prozesskontext geschehen kann. Denn dieser erfordert ja schon einen funktionierenden Objektmanager.

Diese Arbeit konzentriert sich auf die Abläufe, die unterhalb dieser Prozessebene ablaufen.

Alle ausgeführten Aktionen müssen innerhalb des Prozess-Kontextes von dem Prozess erledigt werden, der die Aktion angestoßen hat. Allerdings darf dieser Kontext den aktuellen Knoten nicht verlassen, während Arbeiten des Objektmanagers ausgeführt werden. Wenn Aktionen von außen (anderen Knoten) angestoßen werden, müssen diese innerhalb des Interrupt-Kontextes behandelt werden.

In diesem Fall sind keine Fernaufrufe mehr möglich, da es keinen eigenen Aktivitätsträger für die Verwaltungstätigkeiten gibt.

Also muss auf nachrichtenbasierte Kommunikation zurückgegriffen werden. Um jedoch den Vorteil nicht zu verlieren, dass die auf dem anderen Knoten auszuführende Aktion direkt angeben werden kann, wird diese hier so erweitert, dass eine Mischung aus nachrichten- und fernaufrufbasierter Kommunikation möglich ist.

Dem entfernten Knoten wird eine Nachricht geschickt, die jedoch nicht die Beschreibung eines Ereignisses, sondern einer Aktion enthält. Wenn der entfernte Knoten diese Nachricht erhält, ruft er diese Aktion (eine Funktion) direkt in dem Interrupt-Kontext auf, in dem die Nachricht empfangen wurde. Auf dem sendenden Knoten wird der dort benutzte Aktivitätsträger nicht beeinflusst, sondern arbeitet sofort nach dem Senden weiter.

Dieser Ansatz erlaubt es, sowohl endliche Automaten als auch eine den Fernaufrufen ähnliche Form der Kommunikation zu verwenden.

Die auf dem Zielrechner aufgerufene Funktion kann ihren Aufruf als Ereignis interpretieren und den Zustand eines Automaten ändern.

Wenn der sendende Knoten über einen Prozess-Kontext verfügt, kann auch ein Fernaufruf simuliert werden. Der aufrufende Prozess blockiert sich dabei sofort nach dem Senden. Die auf dem entfernten Knoten aktivierte Funktion erledigt die gewünschte Arbeit und schickt danach eine weitere Nachricht zurück an den Ursprungs-Knoten, die das Ergebnis beinhaltet und den schlafenden Prozess wieder weiter laufen lässt. Dies ist kein echter Fernaufruf, da hierbei während der Ausführung des Funktionsaufrufs nicht mehr der ursprüngliche Prozess-Kontext zur Verfügung steht, sondern nur noch der Interrupt-Kontext.

2.2 Anforderungen

Das Kommunikationsmodul muss gegenüber den Unzulänglichkeiten der vorhandenen Netzwerk-Hardware fehlertolerant sein.

Dies beinhaltet vor allem, dass die Kommunikation zwischen den Knoten nicht gestört werden darf, wenn Pakete doppelt oder gar nicht am Ziel ankommen.

Das Kommunikationsmodul muss absolut zuverlässig sein: Wenn eine Nachricht abgeschickt wird, muss diese auch auf jeden Fall auf dem Zielknoten ankommen.

An verschiedenen Stellen ist es nötig, alle Knoten über ein bestimmtes Ereignis informieren zu können. Hierzu ist es sinnvoll, einen Mechanismus in das Kommunikationsmodul einzubauen, der es erlaubt, eine Nachricht an alle verfügbaren Knoten zu schicken. Dies wird als Broadcast bezeichnet.

Auch muss es auch möglich sein, Ereignisse auf verschiedenen Knoten zeitlich einordnen zu können.

2.3 Implementation

Beim Aufbau eines solchen Kommunikationsmoduls ist es sinnvoll, die einzelnen Aufgaben in verschiedene Schichten zu untergliedern.

Die Nachrichten durchlaufen beim Senden und Empfangen die einzelnen Schichten. Jede Schicht bearbeitet einen Teil der Nachricht und überlässt den Rest der nachfolgenden Schicht (siehe dazu Abbildung 1).

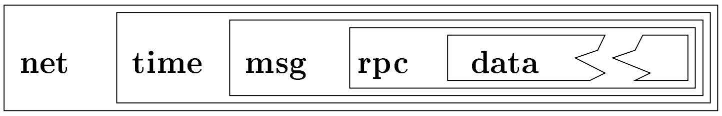

Abbildung 1: Aufbau der verschickten Pakete

2.3.1 Netzwerkschicht

Diese Schicht (`net`) ist für das eigentliche Senden und Empfangen von Nachrichten zwischen den Knoten verantwortlich.

Jeder Knoten hat eine eindeutige Nummer, über die er im Cluster identifiziert werden kann. Aufgabe der Netzwerkschicht ist es, diese Abstraktion zur Verfügung zu stellen, indem die Knotennummer gegen die eigentliche Netzwerkadresse getauscht wird. Dann wird das Paket der Netzwerkhardware übergeben.

Die Netzwerkadressen können sich im Betrieb ändern, wenn ein Knoten durch einen anderen ersetzt wird. Daher wird die Verwaltung dieser Daten auf ein anderes Modul übertragen, den Knotenmanager, der später beschrieben wird.

2.3.2 Zeitsynchronisation

In verteilten Systemen gibt es keine globale Uhr. Damit man Zeitstempel verschicken kann, die dennoch vergleichbar sind, muss eine logische Uhr eingeführt werden, wie sie von Lamport [6] beschrieben wurde.

Dazu verschickt die `time`-Schicht zu jedem Paket noch die aktuelle Zeit auf dem sendenden Knoten. Wenn beim Empfangen einer Nachricht diese einen Zeitstempel trägt, der neuer ist als die aktuelle Zeit auf dem Knoten, wird die Uhr dort entsprechend vorgestellt.

Zurückgestellt wird die Uhr nie, damit ein Zeitpunkt nicht zweimal durchlaufen wird.

2.3.3 Nachrichtenschicht

Leider kann es bei Netzwerken jederzeit passieren, dass einzelne Pakete verloren gehen, oder dass diese durch einen Fehler doppelt empfangen werden. Damit sich der Fehler nicht auf das Verhalten des Systems einwirkt, müssen diese Fälle abgefangen werden. Um die dabei benötigten Algorithmen nicht überall einbauen zu müssen, sollte dies möglichst tief, in dem Kommunikationsmodul erfolgen.

Aufgabe der Nachrichtenschicht (`msg`) ist es, dafür zu sorgen dass jede Nachricht an dem Zielrechner genau einmal ankommt.

Dazu verwendet es ein ähnliches Verfahren, wie es auch in dem Protokoll RDP [10] oder TCP zu finden ist.

Zwischen allen Knoten werden Verbindungen aufgebaut. Auf diesen Verbindungen bekommen alle Pakete eine eindeutige Nummer, die fortlaufend hochgezählt wird. Dadurch kann der empfangene Knoten sichergehen, dass er die Pakete in der richtigen Reihenfolge bekommt. Pakete mit falscher Nummer werden einfach ignoriert. Alle ankommenden Pakete (auch die ignorierten) werden bestätigt. Kommt beim sendenden Knoten keine Bestätigung, oder eine mit einer falschen Nummer an, so sendet er sein Paket noch einmal.

Dadurch können folgende Fehler der Netzwerkhardware ausgeglichen werden:

1. Ursprüngliches Paket geht verloren: Wenn das nächste Paket am Zielknoten ankommt, wird diese ignoriert, aber bestätigt. Der sendende Knoten stellt fest, dass eine Bestätigung fehlt, und sendet das verloren gegangene Paket noch einmal.

2. Ursprüngliches Paket kommt doppelt an: Das zweite Paket ist außerhalb der Reihenfolge und wird daher am Zielknoten ignoriert, aber auch bestätigt. Diese zweite Bestätigung wird auf dem sendenden Knoten ignoriert. Alle weiteren Nachrichten kommen normal an.

3. Bestätigung geht verloren: Für den sendenden Knoten sieht dies zuerst so aus, als ob sein Paket verloren gegangen ist. Demnach wird er es noch einmal senden. Auf dem Zielknoten ist dieses zweite Paket dann außerhalb der Reihenfolge und wird ignoriert, aber wiederrum bestätigt. Diese zweite Bestätigung sieht der erste Knoten dann als gültige Bestätigung.

4. Bestätigung kommt doppelt an: Die zweite Bestätigung wird einfach ignoriert.

5. Pakete kommen in falscher Reihenfolge an: Die Verzögerung eines Pakets wird als dessen Verlust gedeutet und entsprechend noch einmal gesendet. Wenn es dann später doch noch ankommt, wird es als doppelt empfangenes Paket gedeutet und wie oben beschrieben ignoriert.

Ein Prozess kann sich, wenn gewünscht, auch während dem Senden blockieren, bis die Bestätigung des Pakets angekommen ist. Auch der Aufruf einer Callback-Funktion ist möglich (etwa für Senden vom Interrupt-Kontext aus).

In dieser Schicht werden auch Broadcasts behandelt. Es soll möglich sein, auf die Bestätigung eines Broadcasts zu warten (Bestätigung von *allen* Knoten). Dies ist effizient nur in der Nachrichtenschicht möglich. In der vorliegenden Implementation werden Broadcasts durch aufeinanderfolgende Nachrichten an die einzelnen Knoten realisiert. Es werden erst alle Nachrichten verschickt und danach gewartet, bis die Bestätigungen von allen Knoten eingegangen sind.

Zu jeder Nachricht die geschickt werden soll, wird eine eigene kleine Verwaltungsstruktur angelegt, in der gespeichert wird, ob noch Bestätigungen ausbleiben und was passieren soll, wenn die Nachricht erfolgreich an alle Knoten geschickt wurde.

2.3.4 Funktionsaufrufe

Ein Funktions-Zeiger und Parameter in der Nachricht wird von der `rpc`-Schicht benutzt, auf dem empfangenden Rechner eine Funktion aufzurufen.

Da direkt Funktions-Zeiger, und keine Funktions-Ids (etwa wie bei CORBA), benutzt werden, muss auf allen Knoten die *exakt* gleiche Speicherbelegung vorhanden sein. Aber dies wird ohnehin schon benötigt, da beim Versenden von Prozessen ja auch der aktuelle Programmzähler, etc. mitgeschickt wird, und dies auf dem Zielknoten wieder gültig sein muss.

Eventuelle Parameter können mit der Nachricht auch mittransportiert werden. Diese werden jedoch in keiner Weise interpretiert und einfach als Binärdaten mitgeschickt. Um das Ein- und Auspacken der verschiedenen Parameter muss sich die sendende und empfangende Funktion selber kümmern. Dies ist jedoch kein großer Aufwand, die einzelnen Parameter können einfach hintereinander in einen dafür angelegten Speicherbereich kopiert werden.

Diese Schicht ist also sehr viel einfacher gehalten, als RPC (Remote Procedure Call) Schichten in anderen verteilten Umgebungen, die häufig auch Plattformunabhängigkeit anbieten.

Dies ist die oberste Schicht dieses Kommunikationmoduls. Alle Netzwerkoperationen, die zum Verschicken von Objekten nötig ist, können hierdurch zur Verfügung gestellt werden.

3 Objektmanager

Der Objektmanager muss dafür sorgen, dass alle vom Betriebssystem zur Verfügung gestellten Ressourcen unabhängig von dem gerade verwendeten Knoten benutzbar sind. Erst dadurch kann ein für Anwendungsprogramme transparenter Cluster aufgebaut werden.

3.1 Anforderungen

All die verschiedenen Ressourcen sollen effizient verwaltet werden. Dazu ist ein sehr flexibles System nötig.

Damit der Objektmanager möglichst schlank implementiert werden kann, muss er so aufgebaut sein, dass er möglichst wenig von den verschiedenen Objekten wissen muss. Alle Spezialisierung auf die verschiedenen Ressourcen sollte in die Schichten, die den Objektmanager verwenden, verlagert werden.

Der Objektmanager muss dafür sorgen, dass die Objekte jederzeit erreichbar sind. Vor allem dürfen keine Objekte verloren gehen.

Um Inkonsistenzen in den Objekten zu verhindern, muss sichergestellt werden, dass ein Objekt nicht von verschiedenen Knoten zur gleichen Zeit verändert wird.

3.2 exportiertes Interface

Damit das Betriebssystem Objekte als Abstraktion für Ressourcen verwenden kann, muss der Objektmanager die Funktionen zur Verfügung stellen, mit denen auf diese Objekte zugegriffen werden kann.

Für den Objektmanager sind Objekte einfach nur Speicherbereiche von fester Größe. Der Inhalt ist für den Objektmanager vollkommen transparent, er sorgt nur dafür, dass dieser Inhalt auf allen Knoten benutzt werden kann.

Um auf unterschiedliche Anforderungen einzelner Objekte eingehen zu können, werden die Objekte in verschiedene Typen aufgeteilt. Jedes Objekt gehört genau einem Objekttyp an. Allen Objekttypen wird eine Nummer zugewiesen, über die sie unterschieden werden können.

Identifiziert werden die Objekte über ihren Typ und eine eindeutige Nummer, die hier als Id bezeichnet wird.

`obj_create`: Mit dieser Funktion wird ein neues Objekt mit dem angegebenen Typ und Id angelegt. Diese Id muss im ganzen Cluster eindeutig sein. Der Objektmanager muss sich darauf verlassen können, dass die ihm übergebenen Ids gültig sind, und keine doppelt vergeben werden. Um bei nur lokal verwendeten Objekten keine Netzlast zu verursachen, können die Ids nicht noch einmal extra überprüft werden. Auf das effiziente Generieren solcher eindeutigen Ids soll hier nicht weiter eingegangen werden. Als einfachste Möglichkeit können diese etwa aus der Knotennummer und einer laufenden Nummer generiert werden.

Die Funktion gibt einen Speicherbereich zurück, der ab jetzt unter der angegebenen Id ansprechbar ist. Dieser Speicherbereich kann sofort verwendet werden.

`obj_destroy`: Um ein nicht mehr benötigtes Objekt aus dem Cluster zu entfernen, gibt es diese Funktion. Der von dem Objekt belegte Speicher wird von ihr freigegeben. Auch die Id kann nach dem Freigeben des Objektes für ein neues Objekt wiederverwendet werden. Um das Objekt zerstören zu können, muss ein Knoten zuerst sicher gehen, dass er das Objekt auch verändern darf und dieses nicht noch von anderen Knoten verwendet wird.

`obj_enter`: Diese Funktion macht ein Objekt auf dem Knoten gültig und sollte aufgerufen werden, sobald das Objekt initialisiert ist und somit einen konsistenten Zustand erreicht hat.

`obj_leave`: Dies ist das Gegenstück zu `obj_enter`. Nach Aufruf dieser Funktion gilt das Objekt als nicht mehr gültig. Nach eventuellen weiteren Aufräumarbeiten sollte das Objekt im Anschluss mit `obj_destroy` zerstört werden.

`obj_lock`: Um Konsistenz in den Objekten gewähren zu können, muss man verhindern, dass mehrere Knoten ein und das gleiche Objekt zur gleichen Zeit verändern. Auch muss verhindern werden, dass andere Knoten Daten aus einem Objekt lesen, das gerade verändert wird, da sie sonst eventuell inkonsistente Zwischenergebnisse bekommen könnten. Dies wird mit Sperren erreicht. Bevor auf ein Objekt zugegriffen werden kann, muss erst `obj_lock` aufrufen werden. Diese Funktion wartet, bis der Zugriff auf das Objekt erfolgen kann, und sperrt daraufhin das Objekt entsprechend.

Über einen Parameter kann eingestellt werden, ob lesend oder schreibend auf das Objekt zugegriffen werden soll. Schreibzugriffe sind in der

Regel teurer als Lesezugriffe und sollten vermieden werden, wenn dies nicht wirklich notwendig ist.

Wenn notwendig wird das gewünschte Objekt auf den eigenen Knoten transferiert.

Sobald der Zugriff möglich ist, gibt `obj_lock` dann den Speicherbereich des Objektes zurück.

`obj_unlock`: Dies ist das Gegenstück zu `obj_lock`. Eine vorher gesetzte Sperre wird damit aufgehoben, so dass andere Prozesse wieder auf das Objekt zugreifen dürfen. Dies sollte passieren, sobald das Objekt nicht mehr benötigt wird. Wenn das Objekt verändert wurde, muss es sich zu diesem Zeitpunkt wieder in einem konsistentem Zustand befinden.

Nach Aufruf dieser Funktion darf nicht mehr auf den von `obj_lock` gelieferten Speicherbereich zugegriffen werden.

`obj_move`: Mit dieser Funktion kann man den Transport eines Objektes auf einen anderen Knoten veranlassen. Die Funktion kehrt sofort zurück, das Objekt wird dann später verschoben, sobald dies möglich ist (auf dem Ursprungsknoten nicht mehr verwendet wird).

Soll der gerade aktive Prozess verschoben werden, wartet die Funktion jedoch, bis er auf dem anderen Rechner angekommen ist. Dies wird in dem Systemcall `do_migrate` verwendet, mit dem man gezielt auswählen kann, auf welchem Knoten ein Prozess laufen soll.

Diese Funktion braucht eigentlich nur selten selber aufgerufen werden, der Objektmanager sorgt selber dafür, dass sich die Objekte immer auf dem richtigen Knoten befinden. Vor allem ist darauf zu achten, nie ein Objekt explizit auf einen Knoten zu verschieben, wenn dieser gerade herunter gefahren wird.

`obj_migrate_to`: Wenn man auf ein anderes Objekt zugreifen will, ist es oft nicht erwünscht, dieses Objekt dafür von einem anderen Knoten hertransportieren zu müssen.

In manchen Fällen, ist es leichter, wenn der verändernde Prozess stattdessen zu dem Objekt hinmigriert. Dies ist mit dieser Funktion möglich.

Bis auf den Unterschied, dass hier der eigene Prozess verschoben wird, ist diese Funktion semantisch identisch zu `obj_lock`. Wenn das Objekt lokal vorhanden ist, verhält sie sich auch exakt genauso.

Die einzige Voraussetzung, die ein Objekt erfüllen muss, um mit diesem Interface verwaltet werden zu können, ist, dass es durch einem zusammenhängenden Speicherbereich endlicher Größe repräsentiert werden kann.

Dies ist allen wichtigen Ressourcen in einem Betriebssystem der Fall, so dass sich dieser Ansatz hier gut eignet.

Dadurch ist auch sichergestellt, dass man den Objektmanager durch einen leistungsfähigeren austauschen kann, ohne dass dabei darauf zugreifenden Module umstrukturiert werden müssen.

3.3 Verwendung

Damit man sich von der Verwendung dieser Funktionen ein Bild machen kann, soll hier ihr Einsatz kurz mit einigen Beispielen demonstriert werden.

3.3.1 Erstellen von Objekten

Mit `obj_create` wird ein neues Objekt angelegt. Dieses ist für den erzeugenden Knoten sofort schreibbar (so als ob `obj_lock` aufgerufen wurde), und sollte sofort initialisiert werden, etwa indem es mit Nullen gefüllt wird. Ist dies erfolgt, kann das Objekt als gültig erklärt werden, und dann per `obj_unlock` anderen Prozessen zur Verfügung gestellt werden.

```
o = obj_create(type, id);
// initialize o
obj_enter(type, id, WRITE);
obj_unlock(type, id, WRITE);
```

3.3.2 Freigeben von Objekten

Das Zerstören eines Objekt wird als eine Schreiboperation betrachtet. Dadurch wird ausgeschlossen, dass noch andere Knoten auf das Objekt zugreifen, während es vernichtet wird.

```
obj_lock(type, id, WRITE);
obj_leave(type, id, WRITE);
obj_destroy(type, id);
```

3.3.3 Aufrufen von `obj_enter` und `obj_leave`

Diese Funktionen sind eigentlich nur bei manchen Objekten nötig. Zum Beispiel bestimmen sie, welche Prozess-Objekte sich in der Prozess-Liste befinden. Hier ist der korrekte Einsatz dieser Funktionen elementar wichtig.

Bei den meisten anderen Objekten bewirkt er jedoch überhaupt nichts und könnte daher aus Geschwindigkeitsgründen entfallen. Dies ist jedoch nicht zu empfehlen, eventuell werden die Objekte später einmal so verändert, dass diese Funktionen doch gebraucht werden.

3.3.4 Lesen von einem Objekt

Jeder Objektzugriff muss mit `obj_lock` und `obj_unlock` Funktionen eingeschlossen werden.

```
o = obj_lock(type, id, READ);
ret = o->count;
obj_unlock(type, id, READ);
return ret;
```

3.3.5 Verändern von Objekten

Prinzipiell geschieht dies genauso wie ein Lesezugriff, nur muss per `obj_lock` Schreibzugriff verlangt werden.

```
o = obj_lock(type, id, WRITE);
o->count++;
obj_unlock(type, id, WRITE);
```

3.3.6 Verschachtelte Sperren

An vielen Stellen wäre folgende Konstruktion wünschenswert:

```
o = obj_lock(type, id, READ);
// read from o
// ...
if(need_to_write) {
  o = obj_lock(type, id, WRITE);
  // write to o
  obj_unlock(type, id, WRITE);
}
obj_unlock(type, id, READ);
```

Soweit möglich, sind Lesesperren den Schreibsperren immer vorzuziehen. Also muss versucht werden, Objekte wirklich nur dann schreibbar zu sperren, wenn dies nötig ist. Solche verschachtelten Sperren sind aber leider nicht möglich. Man stelle sich zwei Knoten vor, die zur gleichen Zeit obiges Programmstück ausführen wollen. Keiner der beiden kann das Schreiblock betreten, da der andere Knoten das Objekt gerade noch lesbar gesperrt hat (die eigene Lesesperre könnte hier ja ignoriert werden). Um diesen Deadlock zu vermeiden, muss es möglich sein, dass vor dem Betreten des Schreiblocks ein anderer Knoten dieses Objekt verändern darf (damit dieser das Objekt dann auch freigeben kann). Dies verbietet sich jedoch mit einer gehaltenen Lese-Sperre.

Fazit: man kann vorhandene Lese-Sperren nicht ohne weiteres in eine Schreib-Sperre umwandeln. Vorher muss man immer die Sperre aufheben. Ist also abzusehen, dass ein Objekt geschrieben werden muss, muss das Objekt auch von Anfang an schreibbar gesperrt werden. Ansonsten riskiert man, das sich der Inhalt des Objektes ändert.

3.3.7 Fernaufrufe

Oft sollen Objekte auch innerhalb von einem Fernaufruf bearbeitet werden. Dies ist vor allem bei Objekten sinnvoll, bei denen das Verschieben sehr teuer oder gerade nicht möglich ist. In diesem Fall migriert der bearbeitende Prozess zu dem Objekt hin, so dass das Objekt nicht selber verschoben werden braucht.

Hierfür ist `obj_migrate_to` eine bequeme Möglichkeit, da diese atomar den eigenen Prozess zu dem Objekt bringt, und gleichzeitig das Objekt zum Lesen oder Schreiben sperren kann.

```
int origin = myself;
o = obj_migrate_to(type, id, WRITE);
// now we arrived at the object
// start of Remote Procedure Call
o->count++;
// end of RPC
obj_unlock(type, id, WRITE);
do_migrate(origin);
```

3.4 Vorgehen

3.4.1 Konsistenz und Effizienz

Um zu verhindern, dass ein Objekt in einen inkonsistenten Zustand kommt, darf es nie an mehreren Stellen gleichzeitig verändert werden.

Dies wird dadurch sicher gestellt, dass es von jedem Objekt zu jedem Zeitpunkt immer nur ein Original gibt. Der Knoten, auf dem sich dieses Original befindet, wird Master-Knoten für dieses Objekt genannt. Nur auf dem Master-Knoten dürfen Objekte verändert werden.

Wenn ein anderer Knoten ein Objekt verändern will, muss er sich erst das Original zu diesem Objekt beschaffen, und somit zu dem Master für das zu verändernde Objekt werden.

Um bei häufig auftretenden Leseoperationen von verschiedenen Knoten das betreffende Objekt nicht jedesmal transportieren zu müssen, können Kopien eines Objektes auf verschiedenen Knoten angelegt werden. Diese Lese-Kopien dürfen jedoch nicht verändert werden, dies ist nur bei dem Original des Objektes erlaubt.

Auch müssen diese Lesekopien freigeben werden, wenn das Original sich verändert, ansonsten würden lesende Knoten veraltete Informationen bekommen können. Dazu muss sich der Master für das Objekt merken, welche Knoten eine Lesekopie besitzen, und all diese Knoten benachrichtigen, bevor das Original verändert werden darf.

Will ein Knoten ein Objekt verändern, hat aber bereits eine gültige Lesekopie, brauchen die Daten des Objektes nicht noch einmal übertragen werden. Es müssen nur alle restlichen Kopien gelöscht werden, dann kann der schreibende Knoten Master für dieses Objekt werden. Kessler und Livny [4] haben ein ähnliches Verfahren vorgeschlagen, bei dem jedoch noch Sequenznummern benötigt werden, auf die hier wegen der strengen Koherenz verzichtet werden kann.

Netzwerktransfers der Objektdaten werden nur dann vorgenommen, wenn dies wirklich absolut nötig ist, so dass das Gesamtsystem so effizient wie möglich arbeiten kann.

3.4.2 Wiederfinden von Objekten

Wenn es nur ein Original eines Objektes gibt, laufen alle Operationen von diesem Objekt über den Master-Knoten. Um aber Lese- oder Schreibrechte auf das Objekt erlangen zu können, muss ein Knoten erst einmal wissen, wer der Master für dieses Objekt ist.

Dabei sollen Broadcasts – das Senden einer Nachricht an alle Knoten – möglichst vermieden werden. Denn diese belasten das Netz sehr stark, und die Nachricht muss von allen Knoten verarbeitet werden.

Um so wenige Broadcasts wie möglich verschicken zu müssen, verbietet es sich auch, beim Anlegen oder Verschieben von Objekten alle Knoten zu informieren. Damit wüsste zwar jeder Knoten, wo sich alle Objekte befinden, aber die Netzbelastung wäre viel zu hoch. Auch werden viele Objekte oft auch nur von wenigen Knoten benötigt, es müssen also bei weitem nicht immer alle Knoten Kenntnis vom Gesamtzustand besitzen.

Wenn ein Knoten den Master finden will, bleibt ihm aber letzten Endes nichts anderes übrig, als alle Knoten zu fragen, und darauf zu warten, dass der Master antwortet. Um diese Broadcasts aber möglichst zu vermeiden, speichern die Knoten den letzten ihnen bekannten Aufenthaltsort eines Objektes. Dieser wird hier als Hint bezeichnet.

Dieser Hint wird so oft wie möglich aktualisiert. Wann immer Kontakt mit dem Master-Knoten aufgenommen werden soll, wird zuerst der Hint des betreffenden Objekts benutzt, und die Anfrage an den dort gespeicherten Knoten geschickt.

In vielen Fällen ist dies schon der gewünschte Knoten. Wenn nicht, hatte dieser Knoten auf jeden Fall auch Kontakt mit dem Objekt und weiß besser, wer der richtige Master ist. Die Anfrage kann so weiter geschickt werden, bis sie den Master-Knoten erreicht hat.

Oft führt dieses Vorgehen sehr schnell zum Erfolg. Nur wenn ein Knoten zum ersten Mal Kontakt zu einem Objekt aufnimmt, oder wenn der Hint schon sehr veraltet ist, muss ein Knoten dieses mit einem Broadcast suchen.

Dieses Verfahren wird auch bei IVY [3] verwendet um in einem verteilten System gemeinsamen Speicher zur Verfügung stellen zu können.

3.4.3 Bewegen von Objekten

Die Objekte müssen zwischen den einzelnen Knoten hin- und hertransferiert werden können, sei es nun, um eine Lesekopie anzufertigen, oder um das Original eines Objektes auf einen anderen Knoten zu verschieben.

Dabei muss das Objekt auf dem Zielrechner exakt so ankommen, wie es abgeschickt wurde. Eventuell müssen manche Datenstrukturen erst für den Transport vorbereitet werden (verpacken / entpacken).

Da das Vorhandensein eines Objektes unter anderem sehr starken Einfluss auf die Knoten haben kann, darf das Objekt sich zu keinem Zeitpunkt auf mehreren Knoten gleichzeitig befinden. Prozesse könnten sonst unter Umständen auf beiden Knoten weiterlaufen.

Auf der anderen Seite dürfen die Objekte natürlich während dem Transfer auch nicht verschwinden. Es muss jederzeit möglich sein, auf die Objekte zuzugreifen.

Diese Forderungen lassen sich nur erfüllen, wenn die Objekte während dem Transport der Kommunikationsschicht anvertraut werden können. Sobald das Objekt auf den Weg geschickt wurde, wird es von dem Ausgangsknoten entfernt. Dessen Hint zeigt auf den Zielknoten. Der Zielknoten speichert das Objekt, sobald es auf dem Knoten angekommen ist. Bis dahin zeigt sein Hint noch auf den Ursprungsknoten.

Während dem Transport befindet sich ein Objekt also auf keinem Knoten, sondern befindet sich alleine in den Kommunikationsschichten. Wenn ein anderer Knoten auf dieses Objekt zugreifen will, kann sich zu dieser Zeit kein Knoten als zuständig für dieses Objekt melden. Beide Knoten verweisen auf den jeweils anderen, so dass die Anfrage zu dem anderen Knoten weitergeleitet wird. Da die Kommunikationsschichten jedoch die Reihenfolge der Nachrichten beibehalten, kommt die weitergeleitete Nachricht nach dem Objekt auf dem Zielknoten an.

Der Transport eines Objektes wird immer von dem empfangenden Knoten initiiert. Dieser legt schon vorher den für das Objekt benötigten Speicherplatz an, und braucht dann das Objekt bei Empfang nur noch in diesen Speicherbereich entpacken zu lassen.

Wenn ein andere Knoten einen Objekttransport initiiren will, so schickt er dem Zielknoten eine entsprechende Aufforderung. Der Zielknoten sorgt dann dafür, dass das Objekt zum nächstmöglichen Zeitpunkt auf dem Knoten ankommt. Somit sind bis zu drei Nachrichten nötig, um ein Objekt zu transportieren. Dies ist nötig, da sich der Zielknoten erst darauf einstellen muss, ein neues Objekt zu erhalten.

3.5 Objekttypen

Auch wenn sich der Objektmanager transparent mit den Objekten umgehen muss und nichts von deren semantischer Bedeutung weiß, muss bei vielen Operationen auf die Besonderheiten der einzelnen Objekttypen eingegangen werden.

Zum Beispiel muss der Taskmanager alle auf einem Knoten vorhandenen Prozess in eine lokale Liste eintragen. Diese Liste muss immer aktualisiert werden, wenn Objekte auf einem Knoten ankommen oder von dort verschwinden.

Diese Funktionalität in dem Objektmanager zu implementieren, macht wenig Sinn. Sie gehört eher in die Module, die die betreffenden Objekte auch verwenden, in diesem Fall in den Taskmanager.

Es muss also eine Schnittstelle zur Verfügung gestellt werden, über die der Objektmanager die zu dem jeweiligen Objekttyp passende Implementierung von bestimmten Aktionen aufrufen kann.

Nicht immer muss für die einzelnen Objekttypen eine angepasste Implementation zur Verfügung gestellt werden, oft reicht auch eine vom Objektmanager bereitgestellte Standard-Implementation.

Dieses Vorgehen ist ähnlich dem Vererben bei objektorientierten Programmiersprachen. Der Objektmanager stellt eine Klasse mit einer Implementation zur Verfügung. Wenn dies für einen Objekttyp gewünscht ist, kann eine neue Klasse Methoden überschreiben oder aber die alten erben.

Das Interface würde etwa in C++ folgendermassen aussehen:

```
class ObjectType {
        static void* alloc();
        void free();
        void enter(void *mem, int wflag);
        void leave(void *mem, int wflag);
        void change(void *mem, int wflag);
        void pack(void *to, void *from);
        void unpack(void *to, void *from);
        void inval(void *mem);
        static void shutdown();
};
```

Das Ableiten einer Klasse geschieht hier durch das Registrieren eines neuen Objekttyps. Dabei gibt man dessen Typ-Nummer an, sowie die Methoden, die überschrieben werden sollen. Zusätzlich wird hier auch die Größe angegeben, die das Objekt beim Transport einnimmt. Diese ist für alle Objekte eines Typs gleich groß.

Die durch dieses Interface gewonnene Flexibilität wird vor allem dazu genutzt, auf ankommende und den Knoten verlassende Objekte zu reagieren. Die bereitgestellten Methoden werden aufgerufen, um ein neues Objekt zu erzeugen oder zu zerstören (ähnlich dem Konstruktor/Destruktor in einer objektorientierten Programmiersprache), sowie wenn sich der Status eines Objektes auf dem Knoten ändert (nicht vorhanden, Lesekopie, Original). Auch werden sie zum Verpacken der Objekte für den Transport verwendet. Zusätzlich werden die Objekte hierüber informiert, wenn ein anderer Knoten auf das Objekt zugreifen will, und eine lokale Sperre möglichst schnell aufgehoben werden sollte, und wenn der eigene Knoten herunterfährt.

Nötig sind objekttypspezifische Operationen etwa, wenn die Objekte nicht nur in einer globalen, sondern aus Effizienzgründen auch noch in lokalen Datenstrukturen enthalten sind. Dann muss diese Datenstruktur aktualisiert werden, sobald ein Objekt auf dem Knoten ankommt oder von dort verschwindet. Dies wird zum Beispiel bei den Prozessen so realisiert. Hierfür können die `enter` und `leave` Methoden überschrieben werden. Ist es auch noch notwendig, zwischen Lesekopien und Originalen zu unterscheiden, sollte auch noch `change` implementiert werden, um entsprechende Änderungen mitzubekommen.

3.6 Implementation

3.6.1 Verwaltungsstrukturen

Für jedes Objekt, mit dem ein Objektmanager in Berührung kommt, muss eine Datenstruktur angelegt werden, um die Objekte verwalten zu können. In dieser Datenstruktur (`struct obj_mgr`) ist der Zustand des Objektes, sowie ein Zeiger auf den Inhalt des Objektes (`obj_mgr->mem`) enthalten. Auch wenn sich das Objekt auf einem anderen Knoten befindet, bleibt die Verwaltungsstruktur trotzdem bestehen, `obj_mgr->mem` enthält dann `NULL`.

All diese Verwaltungsstrukturen sind nach Id indiziert in einem Hash gespeichert, so dass bei Bedarf schnell darauf zugegriffen werden kann. Dazu werden zwei Funktionen zur Verfügung gestellt, `obj_mgr_lookup` und `obj_mgr_get`. Die erste liefert nur existierende Verwaltungsstrukturen, die zweite legt automatisch eine neue Verwaltungsstruktur an, sollte der Knoten noch nie mit dem angegebenen Objekt in Berührung gekommen sein.

In der Verwaltungsstruktur sind unter anderem folgende Daten gespeichert:

`hint`: Der vermutete Aufenthaltsort des Originals.

`time`: Zeitstempel der letzten `hint`-Aktualisierung.

`cl.rw`: Bestimmt, wie das Objekt auf dem lokalen Knoten verwendet werden darf. (`CL_READ` und/oder `CL_WRITE`)

`cl.cmd`: Während einem Objekttransport wird hier festgehalten, wie weit dieser gerade ist.

`cl.move`: Ist gesetzt, wenn das Original hierher transportiert werden soll.

`cl.inv`: Ist gesetzt, wenn eine Lesekopie gelöscht werden soll.

`cl.rlock / cl.wlock`: Zähler, wie oft das Objekt auf dem lokalen Knoten verwendet wird.

`master[].rw`: Speichert, welcher Knoten wie auf das Objekt zugreifen darf.

`master[].cmd`: Bereits gesendete Befehle vom Master-Knoten werden hier gespeichert. Etwa, wenn Lesekopien auf einem anderen Knoten gelöscht werden sollen.

`wait_queue`: Wenn ein Prozess vom Objektmanager blockiert werden muss, wird er in diese Warteschlange eingetragen.

3.6.2 benutzte Module

Der Objektmanager ist ein sehr grundlegendes Modul, es wird von vielen verschiedenen Schichten verwendet. Dies bedeutet aber auch, dass der Objektmanager all diese Schichten nicht verwenden kann.

Dies betrifft vor allem die Taskschicht. Da diese nicht verwendet werden kann, steht für den Objektmanager kein eigener Prozess zur Verfügung. Damit entfällt auch die Möglichkeit, Daten auf dem Kernel-Stack speichern zu können. Dieser ist für den Objektmanager unter Umständen bei jedem Aufruf ein anderer, etwa wenn verschiedene Prozesse auf Objekte zugreifen wollen.

Die einzigen Schichten, die der Objektmanager wirklich komplett verwenden kann, ist das knotenlokale Speichermanagement, die Kommunikationsschicht, sowie der weiter unten vorgestellte Knotenmanager, mit dem der Objektmanager sehr eng zusammenarbeitet. Innerhalb der Kommunikationsschicht verwendet der Objektmanager nur `rpc`-Aufrufe. Über diese kann er Anfragen an andere Knoten stellen oder diesen Informationen zuschicken. Auf dem empfangenden Knoten wird die `rpc`-Schicht dann die gewünschte Funktion im Objektmanager aufrufen. Hier ist wieder keinerlei Prozess-Kontext garantiert. Oft wird sogar ein solcher Aufruf sogar im Kontext des Idle-Tasks aufgerufen werden, wenn gerade kein lauffähiger Prozess zur Verfügung steht. Diese unbekannte Umgebung wird auch Interrupt-Kontext genannt. Hier muss besonders darauf geachtet werden, dass sich kein Funktionsaufruf blockieren darf.

Innerhalb der vom Objektmanager exportierten Funktionen darf der Objektmanager den aufrufenden Prozess blockieren, wenn dieser auf einen anderen Knoten/Prozess warten muss. Dies wird etwa bei `obj_lock` genutzt. Allerdings kann hier nicht die vom Taskmanager eigentlich hierfür vorgesehene Funktion `interruptible_sleep_on` verwendet werden. Diese würde dem blockierten Prozess erlauben, auf einen anderen Knoten verschoben zu werden. Das Problem ist jedoch, dass der Prozess meistens von einem Interrupt-Kontext wieder aufgeweckt werden soll (wenn das Ereignis, auf das gewartet wird, eintrifft). Aber der Interrupt-Handler darf sich nicht blockieren, die im Objektmanager verwendete `wakeup`-Routine muss also auf jeden Fall atomar und lokal sein.

Extra für den Objektmanager werden die Funktionen `local_sleep_on` und `local_wake_up` benötigt, die zwar nur auf dem lokalen Knoten funktionieren, aber dafür garantieren, dass sich das Aufwecken nicht blockiert.

3.6.3 Asynchrone Programmierung

Wenn sich eine Funktion nicht blockieren darf, hat dies große Auswirkungen auf deren Programmierung.

Dies bedeutet insbesondere, dass keine synchronen Netzwerkoperationen ausgeführt werden dürfen (man darf ja nicht auf deren Bestätigung warten). Stattdessen sollten diese Funktionen möglichst schnell zurückkehren, da sie vom Scheduler nicht unterbrochen werden können und das System in dieser Zeit somit auf nichts reagieren kann.

Wenn man eine Aktion anstoßen will, die jedoch erst ausgeführt werden kann, wenn eine bestimmte Bedingung erfüllt ist, kann man nur festhalten, dass diese Aktion später ausgelöst werden soll. Die Funktion, die diese Aktion anstoßen will, muss sofort zurückkehren, da sie sich nicht blockieren darf und somit nicht selber auf das Eintreten der Bedingung warten kann. Bei jedem Ereignis muss überprüft werden, ob nun alle Bedingungen für noch zu erledigende Aktionen eingetroffen sind. Ist dies der Fall, kann die betreffende Aktion ausgeführt werden.

Somit ist der Objektmanager ähnlich einem Automaten programmiert. Tritt ein Ereignis ein, so ändert sich der Zustand des Objektes, und löst unter Umständen weitere Ereignisse aus. Dieser Zustand wird in der Verwaltungsstruktur des Objektes gespeichert.

Die möglichen Aktionen können zuerst grob in Client- und Masteraktionen unterteilt werden. Zusätzlich gibt es noch einige weitere Aktionen, etwa zum Bestimmen von Hints, die etwas aus diesen beiden Kategorien herausfallen. Diese können jedoch allesamt sofort bearbeitet werden. Hier wird nur auf das Aktualisieren der Hints genauer eingegangen.

3.6.4 Hints

Um schnell den Knoten zu finden, auf dem sich ein Objekt befindet, muss man den vermutlichen Aufenthaltsort in einem Hint speichern.

Damit dieser auch so genau wie möglich ist, muss dieser aktualisiert werden, wann immer dies möglich ist. Bei vielen wichtigen Ereignissen, die ein Objekt betreffen, ist der Master-Knoten bekannt.

Wenn ein Knoten eine Lesekopie des Objektes erhält, kann er davon ausgehen, dass sich das Original auf dem Knoten befindet, der diese Kopie schickte. Solange der Knoten noch diese Kopie besitzt, kann sich an dem Aufenthaltsort des Originals nichts ändern.

Wenn ein Knoten das Original an einen anderen Knoten schickt, weiß er, wo sich das Objekt dann befinden wird.

Auch wenn der Aufenthaltsort nicht immer exakt bekannt ist, kann man anderen Knoten helfen, indem man ihnen in Nachrichten den selber gespeicherten Hint mitschickt. Dies wird etwa genutzt, wenn ein Objekt verschoben werden soll. Der Knoten, auf dem `obj_move` aufgerufen wird, schickt dem Zielknoten die Aufforderung, sich das Objekt zu holen, und schickt den Hint darauf mit, für den Fall dass der Zielknoten das Objekt noch nicht kennt. Dies ist vor allem deshalb sinnvoll, da solche Verschiebe-Operationen oft von Knoten ausgelöst werden, die erst kurz vorher mit dem Objekt in Berührung gekommen sind, oder dieses sogar noch besitzen.

Um bei solchen Ereignissen, bei denen der Hint nicht unbedingt exakt stimmen muss, keinen vorhandenen Hint zu überschreiben, wird bei jedem Hint auch ein Zeitstempel mitgespeichert. Ein Hint wird nur dann verändert, wenn der andere Zeitstempel neuer ist.

Solange ein Knoten schon einmal mit einem Objekt in Berührung gekommen ist, hat er hiermit schon einmal eine gute Idee über dessen Aufenthaltsort.

Sollte ein Knoten trotz alledem noch keine Ahnung von Aufenthaltsort eines Objektes haben, muss er eine Nachricht an alle Knoten schicken, mit der Bitte, ihm den richtigen Hint zu liefern.

Auf diese Anfrage reagieren alle Knoten, die einen Hint zu diesem Objekt gespeichert haben, und senden ihn diesen als Antwort zurück. Jedesmal wenn eine Antwort mit einem Hint eintrifft, der neuer ist als alle vorherigen, kann der Hint aktualisiert werden.

3.6.5 Master-Aktionen

Master-Aktionen sind die Aktionen, die ausgeführt werden müssen, um Anfragen an das Original eines Objektes zu bearbeiten. Die Knoten können zwei verschiedene Anfragen an den Master-Knoten richten: für das Anlegen einer Lesekopie, und für das Übernehmen des Originals.

1. Eine Lesekopie darf erst dann herausgegeben werden, wenn das Objekt gerade nicht mehr verändert wird.

2. Wenn ein anderer Knoten das Original eines Objektes verlangt, muss der Masterknoten zuerst alle Lesekopien einsammeln.

3. Wenn dies von allen betroffenen Knoten bestätigt wurde, und das Objekt auch lokal gerade nicht mehr benutzt wird kann das Original auf einen anderen Knoten transportiert werden.

Um diese Lesekopien zu verwalten, besitzt jeder Master eine Liste, in die alle vergebenen Lesekopien eingetragen werden (`obj_mgr->master[]`). In dieser Liste wird auch vermerkt, wenn ein Knoten schon aufgefordert wurde, seine Lesekopie abzugeben.

Von verschiedenen Knoten können Anfragen an ein Objekt gestellt werden. Diese müssen dann auch in der Reihenfolge bearbeitet werden, in der sie ankommen. Da sie unter Umständen nicht sofort möglich ist, müssen Anfragen in einer FIFO (First In First Out) Warteschlange zwischengespeichert werden.

Die in der Warteschlange gespeicherten Anfragen werden dann nacheinander von der Funktion `obj_master` bearbeitet und aus der Warteschlange entfernt. Ist es im Augenblick nicht möglich, eine Anfrage sofort zu bearbeiten, beendet sich `obj_master`. Angestoßen wird diese Bearbeitung jedesmal, wenn eine neue Anfrage eintrifft sowie bei der Bestätigung eines Knotens, dass dieser seine Lesekopie abgegeben hat.

3.6.6 Weiterleiten von Anfragen

Da Hints nicht immer exakt sind, und Anfragen immer zu den Hints geschickt werden, können auch Anfragen zu Objekten ankommen, die auf dem eigenen Knoten nicht als Original vorliegen. Kommt eine Anfrage für so ein Objekt an, so muss dafür gesorgt werden, dass die Anfrage zu dem Master-Knoten weitergeleitet wird.

Dazu kann man eine Nachricht zurückschicken, die den anfragenden Knoten auffordert, die Anfrage an den richtigen Knoten noch einmal zu stellen. Entweder muss sich der Knoten dann gemerkt haben, welche Anfrage er zu dem Objekt gestellt hat, oder dies muss in der Rückantwort noch einmal explizit enthalten sein. Der Knoten kann dann seinen Hint aktualisieren und die Anfrage neu starten. Zum Aktualisieren des Hints kann die Rückantwort den Hint des anderen Knotens enthalten, der eventuell neuer ist, oder dieser kann gleich neu ermittelt werden.

Eine weitere Möglichkeit besteht darin, die Anfrage so wie sie ist selber an den Master-Knoten zu schicken. Damit dieser dem richtigen Knoten antwortet, muss jedoch der eigentliche Absender explizit in der Nachricht enthalten sein. Mit dieser Alternative kann eine Nachricht eingespart werden, allerdings muss hier der anfragende Knoten damit umgehen können, dass unter Umständen nicht der Knoten antwortet, an den die Anfrage geschickt wurde, sondern ein anderer. Dieses Verfahren wurde auch implementiert.

Wenn die Hints lange Ketten oder gar Zyklen bilden, kann es hier passieren, dass die Anfrage immer wieder von einem Knoten zum nächsten geschickt wird. Die Anfrage erreicht dann erst spät oder überhaupt nicht ihr Ziel. Dieses Problem besteht jedoch bei der ersten Alternative genauso, wenn dort immer der Hint durch den antwortenden Knoten aktualisiert wird.

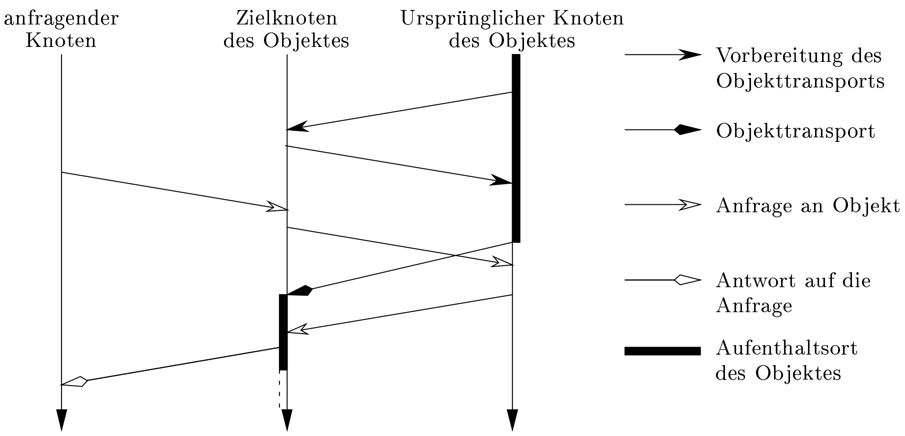

Abbildung 2: Weiterleiten einer Anfrage während einem Objekttransport

Damit die Anfrage trotzdem immer möglichst schnell ihr Ziel erreicht, darf sie nicht ewig weitergeleitet werden. Um dies feststellen zu können, muss mitgezählt werden, durch wie viele Rechner die Anfrage bereits gelaufen ist. Erreicht dieser Wert ein vorher gesetztes Maximum, so müssen die Hints explizit aktualisiert werden, indem alle Knoten nach dem Objekt gefragt werden.

Dies ist vergleichbar mit dem Time-To-Live Feld im IP-Header [9]. Nur dass hier keine Pakete bei Erreichen einer maximalen Lebensdauer gelöscht werden dürfen.

Eintreten kann dies etwa, wenn ein überall bekanntes Objekt von einem Knoten gelöscht wird, und dann auf einem anderen Knoten ein Objekt mit gleicher Id erzeugt wird. Die Hints von allen anderen Knoten zeigen immer noch auf den ersten Knoten, und Anfragen könnten den richtigen Master-Knoten nie erreichen, wenn nur schon vorhandene Hints benutzt werden.

Auch wenn ein Objekt gerade transportiert wird, bilden die Hints einen kleinen Zyklus. Da Nachrichten immer in der gleichen Reihenfolge beim Zielrechner ankommen, werden Anfragen hier maximal zweimal umgeleitet, dann erreicht sie – nach der Ankunft des gesuchten Objektes – den Zielknoten (siehe auch Abbildung 2). Zwei Weiterleitungen sollten also mindestens erlaubt werden, danach kann das Aktualisieren des Hints erzwungen werden.

3.6.7 Client-Aktionen

Client-Aktionen sind all die Aktionen, die von Knoten bearbeitet werden müssen, die nicht Master des Objektes sind. Auch hier betreffen diese wieder Lesekopien und das Verschieben von Originalen.

1. Wenn der Master-Knoten alle Lesekopien zurückziehen möchte, müssen die Clients ihre Kopie sobald wie möglich freigeben und dies dann dem Master melden. Dies ist erst möglich, wenn die Kopie von keinem Prozess mehr verwendet wird.

2. Wenn ein Objekt auf einen anderen Knoten verschoben werden soll, muss dort Speicherbereich zur Verfügung gestellt werden, und dann die Verlagerung des Objektes angestoßen werden.

Wenn ein Knoten die Aufforderung erhält, eine dieser Aktionen auszuführen, wird dies zuerst in der zum Objekt gehörenden Verwaltungsstruktur eingetragen. Hierfür reicht eine einfache Bitmaske aus.

Wird die lokale Kopie eines Objektes nicht mehr verwendet (`obj_unlock`) dann wird `obj_client` aufgerufen.

Diese unternimmt die notwendigen Aktionen, so dies möglich ist.

3.7 Zusammenfassung

Der vorgestellte Objektmanager ist in der Lage, verschiedene Ressourcen unabhängig von deren Aufenthaltsort in einem Betriebssystem zur Verfügung zu stellen. Dabei werden die von Objekten gekapselten Ressourcen automatisch auf die Rechner transferiert, von denen sie verwendet werden.

Auf die verschiedenen Anforderungen der einzelnen Objekttypen kann flexibel eingegangen werden, indem dem Objektmanager hierfür spezialisierte Funktionen mitgegeben werden können.

4 Knotenmanager

Die Aufgabe des Knotenmanagers in diesem verteilten System ist es, die gerade aktiven Knoten zu verwalten.

Zu jeder Zeit muss bekannt sein, welche Knoten gerade verfügbar sind, und wie die einzelnen Knoten erreicht werden können. Da sich diese Informationen ändern können, während das System läuft, wird eine Instanz benötigt, die diese Daten zuverlässig liefern kann. Dies übernimmt der hier beschriebene Knotenmanager.

4.1 Anforderungen

Der Knotenmanager soll folgende Aufgaben bewerkstelligen können:

- Es muss möglich sein, einzelne Knoten des Clusters nach Bedarf hoch- oder herunterzufahren.
- Alle Knoten müssen zu jedem Zeitpunkt eine konsistente Sicht über die vorhandenen Knoten haben. Es darf nicht passieren, dass ein Knoten nur von manchen Knoten aus angesprochen werden kann.

4.2 Status eines Knotens

Der Knotenmanager muss sich bei jedem Knoten merken, ob und wie dieser im Augenblick angesprochen werden kann.

Status eines Knotens kann sein:

`HOST_DOWN`: Derzeit existiert dieser Knoten nicht, oder ist nicht verfügbar.

`HOST_GOING_UP`: Dieser Knoten ist gerade am hochfahren, darf aber noch nicht verwendet werden.

`HOST_UP`: Der Knoten ist betriebsbereit.

`HOST_RETIRE`: Dieser Knoten soll herunter gefahren werden.

`HOST_PASSIVE`: Dieser Knoten ist am herunterfahren und nicht mehr voll einsatzfähig.

Bemerkung: Beim Herunterfahren eines Knotens werden noch ein paar mehr Zwischenzustände benötigt, auf die weiter unten (4.4.5) genauer eingegangen wird.

4.3 Hinzufügen von neuen Knoten

Wenn ein neuer Rechner zur Verfügung steht, soll dieser in den Cluster aufgenommen werden können, während dieser läuft.

Nach dem Aufnehmen des neuen Rechners stehen all seine Ressourcen auch allen anderen Knoten zur Verfügung. Um diese zu nutzen, braucht ein Prozess nur auf diesen Knoten verschoben werden, und kann sofort dessen Rechenleistung und Speicher verwenden. Auch können dann neue Objekte auf diesem Knoten angelegt werden.

Da der Scheduler auf jedem Knoten immer mindestens einen Prozess benötigt, erhält jeder Knoten schon von Anfang an einen Idle-Prozess. Dieser wird auch als Objekt repräsentiert, steht den anderen Knoten jedoch nicht zur Verfügung, da er den Knoten nie verlassen darf.

Andere Aktionen, die ein bootender Rechner normalerweise erledigt, werden hier nicht gebraucht. Der neue Rechner darf/braucht zum Beispiel noch keine Dateisysteme mounten. Sobald er eingefügt wurde, kann er auf den Root-Inode des Dateisystems (der in einem Objekt mit allen bekannten Id gespeichert wird) zugreifen, und kann damit sofort auf alle Dateien im Cluster zugreifen.

Die einzelnen Knoten sollen beim Booten auch kein Wissen über den Cluster benötigen müssen. Ansonsten würde eine Änderung der Clusterkonfiguration ein Neuschreiben der Bootkonfiguration aller Knoten benötigen. Außerdem kann sich der Cluster ja dynamisch verändern, so dass der neu hinzukommende Rechner nie wissen kann, welche Rechner er denn nun wegen seiner Aufnahme fragen kann. Dieser Rechner kann seine Aufnahme also nicht von sich aus anstoßen.

Bootende Rechner fahren ihren Kern hoch und aktivieren ihr Netzwerkinterface. Dann müssen Sie darauf warten, dass der Cluster Kontakt zu ihnen aufnimmt.

4.3.1 Init-Knoten

Diese Vorgehensweise setzt natürlich einen bereits laufenden Cluster voraus, der jedoch erst geschaffen werden muss. Um dem Henne-Ei Problem zu entgehen, benötigt man einen speziellen Knoten (den Init-Knoten), der den Init-Prozess startet. Wie bei einem Ein-Knoten Unix, ist dieser Prozess für das weitere Booten des Rechners bzw. Clusters zuständig.

Dazu muss es ihm möglich sein, andere Rechner in den Cluster aufzunehmen. Hierfür muss ein System-Call zur Verfügung gestellt werden, der den Cluster veranlasst, einen weiteren Knoten aufzunehmen. Beim Aufruf dieses System-Calls müssen dann alle Informationen, die der Cluster über den neuen Knoten benötigt, mitgegeben werden. Dies enthält etwa seine Knotennummer und Netzwerkadresse.

Es ist denkbar, dass der Init-Prozess einen speziellen Daemon startet, der nach weiteren, noch nicht belegten Rechnern im Netz sucht, und diese dann automatisch zum Cluster hinzufügt.

Dies kann dem lokalen Systemadministrator aber auch direkt überlassen werden, so dass neue Rechner immer manuell hinzugefügt werden müssen.

4.3.2 Vorgehen

Das eigentliche Hinzufügen eines neuen Knotens muss in mehreren Schritten vonstatten gehen:

Damit die Netzwerkschicht überhaupt etwas an den neuen Knoten schicken kann, um diesen zu aktivieren, muss erst einmal seine Netzwerkadresse bekannt sein. Jeder Knoten unterhält eine Liste mit allen ihm bekannten Knoten. Zuerst muss der initiierende Knoten den neuen Knoten in diese Liste aufnehmen. Nun kann er ihm eine Nachricht schicken. Diese enthält seine komplette Liste der Knoten sowie die Knotennummern der beiden Partner. Damit ist der neue Knoten komplett initialisiert, es müssen nur noch die restlichen Knoten des Clusters informiert werden.

Dazu wird die Nummer und Adresse des neuen Knotens an alle anderen Knoten verschickt, die diesen dann auch in ihre Liste aufnehmen.

Erst, wenn dies von allen Knoten bestätigt wurde, kann der neue Knoten offiziell angemeldet werden.

Würde dies in einem Schritt geschehen, könnten bereits Prozesse auf den neuen Knoten gelangen, bevor alle restlichen Knoten von dem Vorhandensein des neuen Knotens unterrichtet sind, so dass diese Objekte dann nicht mehr überall erreichbar wären.

4.3.3 Implementation

Zum Aktivieren von Knoten arbeitet der Knotenmanager sehr eng mit den Kommunikationsschichten zusammen, vor allem mit der Nachrichtenschicht.

Schließlich muss die Nachrichtenschicht die Netzwerkadressen der Knoten wissen, wenn etwas zu diesen geschickt werden soll. Da diese Information sich aber ändern kann, wenn ein Knoten herunter- und wieder heraufgefahren wird, und der Knotenmanager für diese Änderungen verantwortlich ist, befindet sich diese Liste im Knotenmanager. Der Netzwerkschicht wird eine Funktion zur Verfügung gestellt, mit der sie lesend auf diese Adressen zugreifen kann (`host_addr`).

Diese Liste kann auf verschiedene Arten manipuliert werden. Es können einzelne Knoten hinzugefügt (`host_add`) oder entfernt (`host_del`) werden. Auch kann diese für den Transport ganz oder teilweise in einen zusammenhängenden Speicherbereich kopiert werden, und bei Bedarf wieder eingelesen werden. Zu jedem Knoten wird noch sein Status gespeichert, der mit `host_change_status` oder `host_change_status_async` (wenn kein Prozess-Kontext vorliegt) geändert werden kann. Dabei wird der Zustand zuerst auf allen anderen Knoten verändert, danach wird auch auf dem lokalen Knoten der Zustand dieses Knotens aktualisiert. All diese Änderungenmöglichkeiten sind jedoch nur vom Knotenmanager selber erlaubt, und werden automatisch beim Hinzufügen/Entfernen von Knoten verwendet.

Der Systemcall `addhost` startet das Hinzufügen eines neuen Knotens. Dazu wird dieser zunächst lokal bekannt gemacht. Der neue Knoten bekommt den Status `HOST_GOING_UP`. Hiermit wird verhindert, dass Objekte auf den noch nicht einsatzfähigen Knoten transferiert werden.

Als erstes muss der neue Knoten den bereits existierenden Cluster kennenlernen. Dies kann nicht über die sonst verwendeten `rpc` Aufrufe geschehen, da diese sofort deren Erhalt über die `msg`-Schicht bestätigen. Dies ist aber noch nicht möglich, da ja erst danach die anderen Knoten bekannt sind. Deshalb wird eine spezielle Funktion zur Verfügung gestellt, die an der `msg`-Schicht vorbei ein Paket schickt (`msg_activate_send`).

Wenn die entfernte Nachrichtenschicht dieses Paket erhält, gibt sie die darin enthaltenen Daten direkt wieder an den dortigen Knotenmanager weiter (dort wird `host_activate_recv` aufgerufen). Dieser konfiguriert mit Hilfe dieser Daten die Verbindungen zu allen Knoten des Clusters. Erst danach ist es möglich, Pakete zu verschicken, und die Nachrichtenschicht kann die Aktivierung des Knotens bestätigen, wodurch die `msg_activate_send` Funktion zurückkehrt.

Der neue Knoten ist nun bereit, von allen Knoten Nachrichten entgegen zu nehmen. Allerdings wissen die übrigen Knoten noch nichts von seiner Existenz, so dass immer noch keine Objekte auf den Knoten gelangen dürfen.

Wurde der Knoten erfolgreich aktiviert, werden nun auch alle anderen Knoten von ihm informiert, immer noch mit Status `HOST_GOING_UP`.

Wenn dies von allen Knoten bestätigt wurde, sind die Kommunikationsschichten von allen Knoten wieder in einem konsistentem Zustand. Ab diesem Zeitpunkt dürfen Objekte auf den neuen Knoten gelangen, also wird der Status des neuen Knotens auf `HOST_UP` gesetzt.

Damit wurde der Knoten erfolgreich in den Cluster aufgenommen. Die

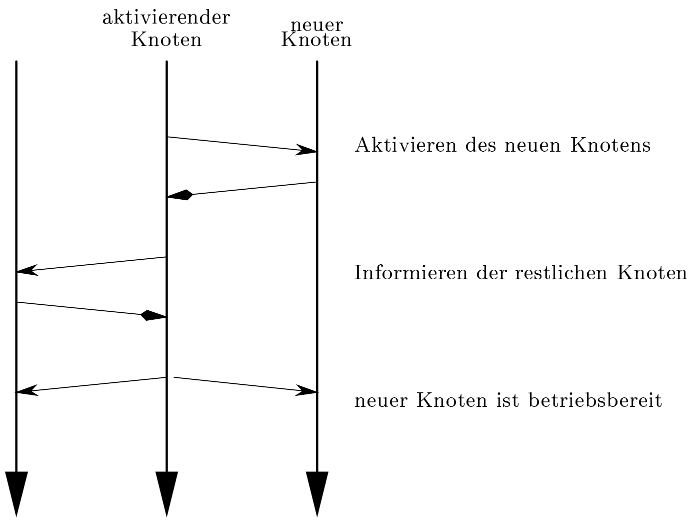

Abbildung 3: Hinzufügen eines neuen Knotens

hier gesendeten Nachrichten sind noch einmal in Abbildung 3 verdeutlicht.

4.4 Herunterfahren von Knoten

Das Hochfahren der Knoten ist für den Betrieb eines Clusters essentiell, aber nicht sonderlich anspruchsvoll.

Um auch große Cluster vernünftig administrieren zu können, muss es jedoch auch möglich sein, einzelne Knoten herunterzufahren, ohne dabei das komplette System abschalten zu müssen.

Um das laufende System dabei nicht zu beeinträchtigen, dürfen die auf dem herunterfahrenden Knoten gespeicherten Objekte natürlich nicht verloren gehen. Schon ein verloren gegangenes Objekt kann das gesamte System zerstören. Man denke dabei nur an den Init-Prozess.

Also müssen alle Objekte dieses Knotens vor dem eigentlichen Abschalten vorher auf andere Knoten verlagert werden. Erst dann kann der Knoten abgeschaltet werden. Das Sichern der Objekte auf andere Knoten kann der Objektmanager übernehmen. Aufgabe des Knotenmanagers ist es, sicher zu gehen, dass der Knoten nicht abgeschaltet werden kann, solange nicht alle Objekte ‚in Sicherheit' sind. Hier muss vor allem darauf geachtet werden, dass sich zu diesem Zeitpunkt keine Objekte mehr auf dem Weg zu dem heruntergefahrenem Knoten befinden, und dass keine Knoten diesen mehr für irgendwelche Anfragen benötigen.

4.4.1 Vorgehen

Das eigentliche Problem beim Herunterfahren ist, dass womöglich noch Objekte gerade auf dem Weg zu dem Knoten sind. Ganz unterbinden kann man die Annahme von neuen Objekten leider auch nicht, da zur Freigabe von dort benutzten Ressourcen eventuell noch andere Objekte nötig sind. (Zum Beispiel eine verteilte Semaphore, die von einem Gerätetreiber benötigt wird, Speicherseiten, etc.)

Wenn man irgendwie sichergehen kann, dass dies nicht mehr notwendig ist, kann man in Ruhe die noch verbliebenen Objekte auf andere Knoten verteilen und den Knoten dann abschalten.

Hierfür ist es sinnvoll, die vorhandenen Objekttypen in zwei Kategorien einteilen: aktive und passive. Als aktive Objekte sollen hier all die Objekte bezeichnet werden, deren Vorhandensein auf einem Knoten andere Objekte anziehen könnte. Dies sind zum Beispiel Prozesse. Gemeint ist hier nicht die Fähigkeit eines Prozesses, sich selber auf einen anderen Knoten zu verschieben, sondern vielmehr, dass andere Objekte von diesem zum Betrieb benötigt werden und von selber zu einem solchen Objekte bewegt werden können. Etwa braucht ein Prozess Speicherseiten, etc, die bei Bedarf geladen werden müssen. Oder es wird ein sterbender Prozess kurz all seine Kinder besuchen, um diese an den Init-Prozess abgeben zu können. Solange also noch ein Prozess auf dem Knoten existiert, kann es jederzeit nötig sein, dass auch andere Objekte dorthin kommen.

Diese Notwendigkeit, dass Objekte zu aktiven Objekten hinmigrieren müssen, wird man nicht verhindern können. Also muss man möglichst schnell alle aktiven Objekte auf andere Knoten verschieben. Dann kann dann sicher sein, dass man sich nur noch um die verbliebenen kümmern muss, und nicht mit neu ankommenden Objekten rechnen muss.

Bevor man beginnt, alle Objekte wegzuschieben, sollte man sich erst um alle aktiven Objekte kümmern, sonst muss man unter Umständen die mit viel Aufwand verteilten Speicherseiten wieder einsammeln, weil diese noch von einem noch laufenden Prozess benötigt werden.

Der Vorgang des Herunterfahrens wird also in verschiedene Phasen unterteilt, die hier mit `HOST_RETIRE` und `HOST_PASSIVE` bezeichnet werden.

`HOST_RETIRE`: Über einen Systemcall kann ein Knoten veranlasst werden, aus dem Zustand `HOST_UP` in den Zustand `HOST_RETIRE` zu gehen. Damit wird das Herunterfahren des Knotens initiiert.

Ab diesem Zeitpunkt dürfen keine Prozesse mehr über `sys_migrate` auf diesen Host verschoben werden. Objekte dürfen jedoch noch zu dort vorhandenen 'aktiven' Objekten hinmigrieren (z.B.: Tasks für Aufräumarbeiten).

Gleichzeitig beginnt der betroffene Knoten, alle auf ihm gespeicherte Objekte loszuwerden. Dazu fängt er mit den aktiven Objekten an.

Sind alle aktiven Objekte verschoben, kann er in den nächsten Zustand wechseln.

`HOST_PASSIVE`: In diesem Zustand dürfen keine Objekte mehr an den Knoten geschickt werden. Es sind keinerlei aktive Objekte mehr auf dem Knoten vorhanden, nur passive Objekte werden noch so lange gespeichert, bis auch diese alle auf andere Knoten transferiert wurden.

Wenn alle noch aktiven Knoten das Herunterfahren bestätigt haben (also keine neuen Objekte mehr schicken) und alle Objekte von dem Knoten weg geschoben wurden, steht dem endgültigen Abschalten nichts mehr im Wege.

Allerdings können noch viele Hints auf den jetzt deaktivierten Knoten zeigen. Diese müssen entweder noch vor Deaktivierung der Netzverbindung oder dann bei Benutzung erneuert werden.

Am einfachsten ist, diese ungültig zu machen. Sobald sie dann später benötigt werden, wird dann per Broadcast nach dem Objekt gefragt. Dadurch wird zwar beim Gebrauch ein Broadcast benötigt, aber um den Hint gleich korrekt zu setzen, wäre auch ein Broadcast nötig. Schließlich müsste der herunter zu fahrende Knoten für jedes verschobene Objekt den Zielknoten an alle anderen Knoten melden. Die entsprechenden Hints erst bei erneuter Benutzung zu erfragen ist bei vielen Objekten sehr viel effizienter.

4.4.2 Geräte-Treiber

Eventuell sind noch Prozesse in Geräte-Treibern blockiert und können deswegen noch nicht sofort verschoben werden.

Alle Geräte-Treiber müssen daher informiert werden, dass der Knoten herunterfährt und deshalb die dort vorhandenen Geräte jetzt nicht mehr verwendet werden können. Die Geräte-Treiber sollten daraufhin alle von ihnen blockierten Prozesse aufwecken und die System-Aufrufe mit einem Fehlerkode (zum Beispiel ENXIO: No such device) beenden.

4.4.3 Verschieben der Objekte

Bei den Objekten, die sich beim Beginn des Herunterfahrens noch auf dem Knoten befanden, kann das Verschieben sofort angestoßen werden.

Probleme gibt es nur, bei Objekten, die im Nachhinein auf den Knoten gelangt sind. Aktive Objekte müssen eventuell extra zurückgeschickt werden. Da in der Host- bzw. Objektmanagerschicht noch keine Prozesse zur Verfügung stehen, um dies im Hintergrund erledigen zu können, muss dies schon sofort bei Ankunft erledigt werden.

Die Objekte werden dabei der Reihe nach auf die noch funktionstüchtigen Knoten aufgeteilt. Diese holen sich dann die Objekte ab.

4.4.4 Bedingungen für einen Zustandsübergang

Wenn alle aktiven Objekte vom Knoten wegbewegt wurden, kann dieser von `HOST_RETIRE` in den Zustand `HOST_PASSIVE` übergehen. Entsprechend in den Zustand `HOST_DOWN`, wenn komplett alle Objekte auf den Rest des Clusters verteilt sind.

Um das Eintreten dieser Situation bemerken zu können, müssen diese Bedingungen sowohl beim Entfernen eines Objektes als auch bei Zustandswechseln eines Knotens überprüft werden.

Dazu muss zuerst einmal die Anzahl der auf diesem Knoten vorhandenen Objekte bekannt sein. Also muss der Objektmanager ständig mitzählen, wie viele Objekte sich auf dem eigenen Knoten befinden. Hier muss aufgepasst werden, dass Objekte, die sich noch auf dem Weg zu diesem Knoten befinden, schon mitgezählt werden. Dies ist möglich, da der Transport eines Objektes letztendlich immer vom Zielknoten aus eingeleitet wird.

Leider reicht es nicht aus, wenn nur der herunterfahrende Knoten weiß, dass dort keine Objekte mehr gespeichert sind. Solange andere Knoten noch davon überzeugt sind, dass sich dort aktive Objekte befinden, kann es passieren, dass diese ein Objekt zu dem vermeintlich noch dort befindlichen aktiven Objekt schicken. (siehe Abbildung 4).

Ähnlich ist es auch mit den passiven Objekten. Solange ein Knoten noch davon überzeugt ist, dass sich ein Objekt auf dem Knoten befindet, muss der betreffende Knoten auch Anfragen nach diesem Objekt beantworten können (wenn auch nur dadurch, dass er die Anfrage unbearbeitet zurückschickt).

Erst wenn sich alle Knoten einig sind, dass sich keine Objekte mehr auf dem betroffenen Knoten befinden, kann dieser einen Zustandsübergang initiieren.

Zu bemerken ist hier noch, dass der Idle-Task, der auf jedem Knoten auch als Objekt repräsentiert, aber nie verschoben wird, nicht mit berücksichtigt werden darf. In Wirklichkeit geschehen solche Zustandswechsel also bei Erreichen von `1` im Objektzähler.

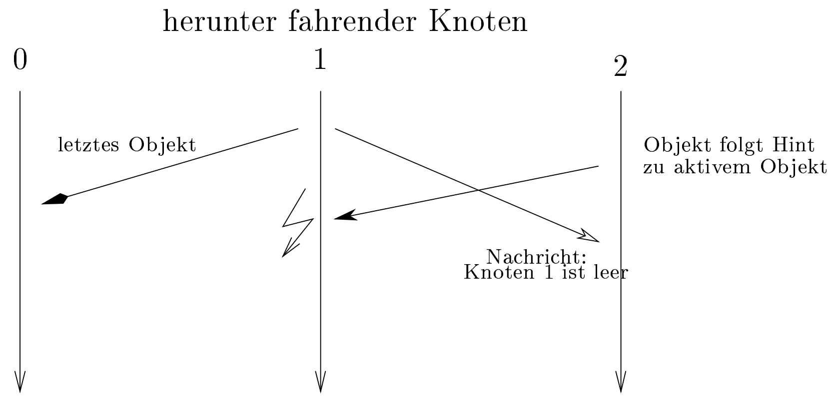

Abbildung 4: Objekt kommt an, obwohl Knoten leer sein sollte

4.4.5 Einigung über bevorstehenden Zustandswechsel

Ein Zustandsübergang ist erst möglich, wenn kein Hint mehr auf den herunterfahrenden Knoten zeigt und somit auch kein Objekt mehr angesprochen wird. Damit die anderen Knoten ihre Hints löschen können, muss man sie informieren, sobald der Knoten leer ist.

Das Problem ist genau dieser Zeitpunkt. Denn bis alle Knoten informiert wurden, können diese noch Objekte auf den herunterfahrenden Knoten schicken. Solange dies aber noch möglich ist, kann der Knoten auch nicht mit Sicherheit von sich behaupten, dass er leer ist.

Die anderen Knoten müssen aber auf jeden Fall informiert werden, zumindest also darüber, dass der Knoten während dem Senden der Nachricht leer war. Beim Empfang dieser Nachricht können dann alle anderen Knoten die Hints, die noch auf diesen Knoten zeigen löschen.

Nur wenn der Empfang dieser Nachricht von allen Knoten bestätigt wurde, und in der Zwischenzeit keine neuen Objekte auf dem herunter zu fahrenden Knoten ankamen, kann sicher in den gewünschten Zustand übergegangen werden.

Kam in der Zwischenzeit ein neues Objekt auf dem Knoten an, muss dieses erst wieder weggeschickt werden, und der Vorgang anschließend wiederholt werden.

Da bei jedem Durchlauf alle auf den Knoten zeigenden Hints gelöscht werden, und nur dann neue auftreten, wenn während dem Informieren ein Objekttransport angestoßen wird, können sich die Knoten sehr schnell darauf einigen, dass der Knoten wirklich komplett leer ist. Somit terminiert dieser Algorithmus recht schnell, auch wenn er theoretisch beliebig lange brauchen kann.

4.4.6 Übergang in nächsten Zustand

Die eigentlichen Zustandswechsel werden von dem Knotenmanager mit Hilfe der Funktion `host_change_status_async` (es liegt hier kein Prozesskontext vor) vorgenommen. Diese benachrichtigt alle Knoten von dem bevorstehenden Wechsel und aktualisiert nach Bestätigung aller Knoten den Zustand des eigenen Knotens.

Durch diese Maßnahmen wird sichergestellt, dass ein einmal erreichter Zustand auch auf jeden Fall sicher ist. Der Knoten kann sicher sein, dass im Zustand `HOST_PASSIVE` kein aktives Objekt mehr auf dem Knoten ist (es wurden ja alle gezählt), keine ein aktives Objekt betreffenden Anfragen an ihn gestellt werden (es wurden alle Hints darauf gelöscht) und kein solches Objekt gerade auf dem Weg zu ihm ist (man darf nur zu aktiven Objekten hinmigrieren, und alle Hints auf diese wurden schon entfernt). Dasselbe gilt für den nächsten Wechsel hin zu `HOST_DOWN`. Hier ist sicher, dass keinerlei Kommunikation dieses Knotens mehr nötig ist, da sich dort keinerlei Objekte mehr befinden.

4.4.7 Abschaltung

Das Herunterfahren eines Knotens ist beendet, wenn dieser den Zustand `HOST_DOWN` erreicht hat.

Dann können alle Netzwerkverbindungen zu diesem gekappt werden. Der Knoten selber kann dann seine Stromversorgung abschalten oder auch einfach rebooten und auf eine erneute Aktivierung durch einen Cluster warten.

4.4.8 Implementierung

Um feststellen zu können, was für Objekte sich noch auf dem Knoten befinden, besitzt jeder Objekttyp einen Zähler, der die gerade auf einem Knoten befindlichen und gerade dorthin transportierten Objekte zählt. Diese Zähler werden von `obj_entering` und `obj_leaving` manipuliert, die aufgerufen werden, wenn Objekte den Knoten betreten beziehungsweise verlassen.

Um das beschriebene Einigungsprotokoll zu implementieren, wird hier der sowieso vorhandene Mechanismus für Zustandswechsel benutzt. Dafür werden noch einige Zwischenzustände eingeführt.

Um die noch auf anderen Knoten vorhandenen Hints zu entfernen, wird ein Zustandsübergang ausgelöst, auf den der Objektmanager reagiert. Da `host_change_status_async` den Zustand zuletzt auf dem eigenen Knoten aktualisiert, weiß man, dass bei Erreichen dieses Zustands die Nachricht von allen Knoten bestätigt wurde.

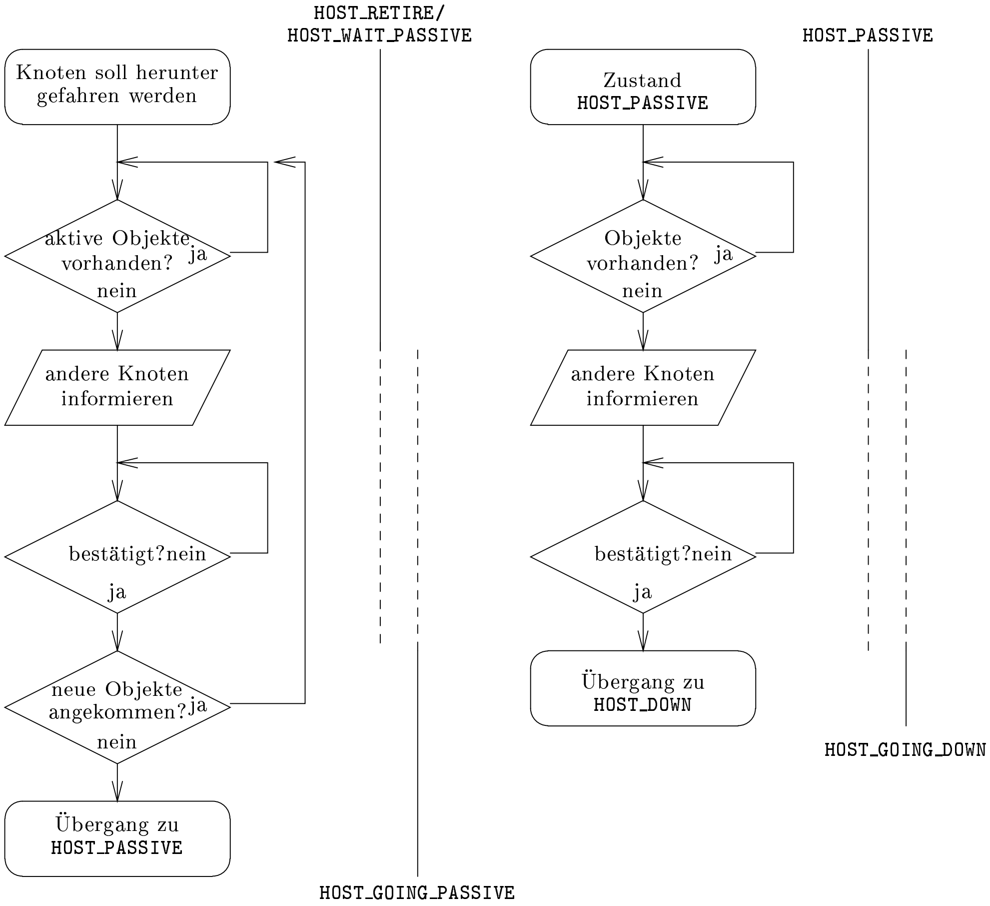

Abbildung 5: Herunterfahren eines Knotens

Die hier verwendeten Zustände sind:

HOST_WAIT_PASSIVE: Während den Pausen zwischen mehreren Durchgängen. Wird nur lokal verwendet, kein anderer Knoten bekommt diesen Zustand zu sehen.

HOST_GOING_PASSIVE: Mit diesem Zustand wird signalisiert, das ein Knoten als leer betrachtet wird.

HOST_PASSIVE: Der zu erreichende Endzustand.

Sobald der Knoten keine aktiven Objekte mehr enthält, initiiert dieser einen Übergang in HOST_GOING_PASSIVE. Dies veranlasst alle anderen Knoten, ihre aktiven Hints (hiermit sind die Hints gemeint, die auf aktive Objekte zeigen) auf den jeweiligen Knoten zu entfernen.

Dabei müssen die Knoten den Zeitstempel des Hints beachten. Wenn der Transport eines Objektes angestoßen wurde, kann dieser zwar erst nach Bestätigung von `HOST_GOING_PASSIVE` – und damit nach dem Löschen der aktiven Hints – zu Ende geführt werden. Aber ein Knoten der einen solchen Transport vorgenommen hat, könnte durchaus einem anderen Knoten, der den `HOST_GOING_PASSIVE` Zustand noch nicht mitbekommen hat, über diesen neu gesetzten Hint informieren. Dieser sollte dann nicht mit entfernt werden.

Wenn alle Knoten den Empfang dieser Nachricht bestätigt haben, und sich der herunter fahrende Knoten auch selber als `HOST_GOING_PASSIVE` markiert, kann er weitere Aktionen einleiten.

Um festzustellen, ob in der Zwischenzeit Objekte auf dem Knoten ankamen, so dass ein weiterer Durchgang notwendig ist, reicht es nicht aus, zu überprüfen, ob sich nun neue aktive Objekte auf dem Knoten befinden. Denn diese könnten eventuell schon wieder auf einen anderen Knoten verschoben worden sein, bevor alle Knoten `HOST_GOING_PASSIVE` bestätigten. In diesem Fall könnten aber auch noch Hints auf den Knoten zeigen.

Vielmehr muss überprüft werden, ob seit der letzten Überprüfung neue Objekte ankamen. Für diesen Zweck gibt es zusätzlich noch einen zweiten Objektzähler, der nur von `obj_entering` hochgezählt wird, jedoch nicht von `obj_leaving` wieder erniedrigt wird. Dieser zweite Zähler wird bei jeder Überprüfung auf Null gesetzt. Wenn dieser Zähler bei der zweiten Überprüfung ungleich Null war, so ist inzwischen ein neues Objekt angekommen und es zeigen eventuell noch Hints auf den Knoten.

Wenn ein neues aktives Objekt auf den Knoten kam, muss der Knoten warten, bis dieses wieder entfernt wurde. Dazu wechselt er in den Zustand `HOST_WAIT_PASSIVE`. Dieser ist in diesem Algorithmus identisch zu `HOST-RETIRE`. Die unterschiedlichen Zustände wurden gewählt, da bei `HOST_RETIRE` noch einige Aktionen angestoßen werden, die hier nicht wieder wiederholt werden müssen. Über diesen Zustandswechsel müssen die anderen Knoten nicht unbedingt informiert werden. Sie müssen erst erfahren, wann der Knoten wieder als leer anzusehen ist.

Wenn während dem Informieren der restlichen Knoten jedoch nichts passierte, das einem Übergang in den Zustand `HOST_PASSIVE` im Wege steht, kann endlich dieser Übergang angestoßen werden.

Der Übergang in den Zustand `HOST_DOWN` ist prinzipiell sehr ähnlich, jedoch hat man hier nicht mehr das Problem, dass unvorhergesehen neue Objekte ankommen. Daher kann auf ein Wiederholen der Prozedur hier von vornherein verzichtet werden.

Es muss hier sichergestellt werden, dass alle Hints auf den Knoten entfernt werden, bevor dieser abgeschaltet wird, um zu verhindern, dass zu diesem Zeitpunkt noch eine Anfrage an den Knoten gestellt wird.

Ferner muss darauf geachtet werden, dass auch alle auf dem herunterfahrenden Knoten vorhandenen Lesekopien entfernt werden. Da diese auf dem Knoten sowieso nicht mehr benötigt werden, kann man entsprechende Eintrage des Masters einfach löschen.

4.5 Gewährleistung der Konsistenz

Während all dieser Operationen die Zustände im ganzen Cluster konsistent zu halten ist eine nicht-triviale Sache. Die vielen parallelen Abläufe müssen alle koordiniert werden.

4.5.1 Einschränkung der Nebenläufigkeit

Die Arbeit des Knotenmanagers wird sehr erschwert, wenn man nebenläufiges Hoch- und Herunterfahren erlaubt.

Die einfachste Lösung besteht darin, dies zu verbieten. Änderungen an der Konfiguration des Systems sind also nur in einem kritischen Abschnitt möglich.

Die damit verbundenen Einschränkungen sind geringer als man auf den ersten Blick vermutet:

Hochfahren des Systems: Beim Starten des Systems müssen alle Knoten nacheinander initialisiert werden. Dies geht jedoch recht schnell und dauert selbst bei großen Systemen nur wenige Sekunden. Dies ist in der Regel vernachlässigbar, verglichen mit der restlichen Boot-Dauer.

Herunterfahren des Systems: Alle Knoten müssen nacheinander heruntergefahren werden. Das Entfernen eines Knotens dauert wesentlich länger als des Einfügen, da erst noch alle dort befindlichen Objekte verschoben werden müssen. Allerdings soll das System ja komplett abgeschaltet werden, so dass der Init-Prozess schon die meisten Prozesse beendet haben sollte. Damit sind auch kaum noch Objekte zum Verschieben vorhanden, so dass das Entfernen der Knoten wieder vernachlässigbar zum Herunterfahren des gesamten Systems sein sollte.

Wartungsarbeiten an einem Rechner: Wenn an der Hardware von einem Knoten gearbeitet werden soll, kann dieser vorher herunter und nach Abschluss der Arbeiten wieder hochgefahren werden. Da hier nur ein Knoten betroffen ist, macht sich der kritische Abschnitt nicht bemerkbar

Not-Aus von Knoten: Wenn die Hardware eines Knotens droht auszufallen, sollte dieser vorher sicherheitshalber heruntergefahren werden, um nicht das gesamte System zu bedrohen. Da sich jedoch noch recht viele Objekte auf diesem Knoten befinden können, dauert dies mitunter recht lange.

Drohen nun mehrere Rechner gleichzeitig auszufallen (z.B. wegen Spannungszusammenbruch), ist es unter Umständen nicht mehr möglich, alle betroffenen Knoten nacheinander herunter zu fahren. Hier macht sich diese Einschränkung sehr stark bemerkbar. Man muss die Rechner auf Hardwareseite besser gegen Ausfall schützen, was teurer ist.

Für die normalen Anwendungsfälle ergibt sich somit aber keine bedeutende Einschränkung.

4.5.2 Zustandsübergänge

In einem verteilten System kann man nie garantieren, dass Zustandsübergänge exakt zur gleichen Zeit eintreten.

Statt dessen muss gewährleistet werden, dass die Reihenfolge, in der diese auf den unterschiedlichen Knoten erfolgen, so gewählt ist, dass das System trotzdem noch in einem konsistentem Zustand bleibt.

Hochfahren eines Knotens: Hier wird der neue Zustand zunächst im neuen Knoten gesetzt, und erst danach auf den bereits laufenden Knoten. Damit ist sichergestellt, dass ein neuer Knoten dem System voll zur Verfügung steht, wenn er dem System bekannt gemacht wird.

Ferner wird das Bekanntmachen (Übergang in Zustand `HOST_UP`) in zwei Schritten erledigt. Zuerst werden die Netzwerk-Schichten aller Knoten informiert (`HOST_GOING_UP`), so dass bidirektionale Kommunikation möglich ist. Erst wenn dies von allen Knoten bestätigt wurde, wird der Knoten als verfügbar markiert. Dadurch wird sichergestellt, dass der neue Knoten, wenn er neue Objekte aufnehmen kann, bereits mit allen Knoten kommunizieren kann.

Herunterfahren eines Knotens: Hier wird in der umgekehrten Reihenfolge vorgegangen. Der herunterfahrende Knoten ändert seinen Zustand erst, wenn der neue Zustand bereits von allen anderen Hosts bestätigt wurde. Auf diese Weise wird dafür gesorgt, dass Nachrichten/Objekte noch von dem Knoten weg gesendet werden können, aber keine mehr zu dem Knoten hin geschickt werden müssen.

Ähnlich wie beim Hochfahren geschieht die ganze Operation in mehreren Schritten, so dass sich die einzelnen Knoten langsam darauf vorbereiten können. Die Einzelheiten hierzu wurden schon im letzten Abschnitt behandelt.

4.6 Zusammenfassung

Erst durch Verwendung eines solchen Knotenmanagers lässt sich ein größerer Cluster von Knoten effizient administrieren.

Das gesamte System ist dadurch sehr leicht skalierbar – ohne dass es dazu heruntergefahren werden muss. Wer mehr Rechenleistung benötigt, kann einfach einen weiteren Rechner dazustellen und diesen aktivieren.

Auch bestehende Rechner können sehr einfach aufgerüstet werden. Dazu wird der betroffene Knoten kurzzeitig deaktiviert. Derweil können der Prozessor, Speicher oder sonstige Ressourcen ausgewechselt und durch bessere ersetzt werden. Nach erfolgreichem Umbau wird der leistungsstärkere Knoten wieder in das System aufgenommen.

Um diese Ziele realisieren zu können, muss sehr eng mit dem Objektmanager zusammengearbeitet werden, der dafür sorgt, dass auf alle Objekte auch nach dem Abschalten eines Rechners noch zugegriffen werden kann.

5 Ausblick

Mit dem vorgestellten Objekt- und Knotenmanager lässt sich ein Cluster aufbauen, in dem allen Anwendungsprogrammen automatisch alle vorhandenen Ressourcen zur Verfügung stehen. Ein aufwendiges, speziell auf Cluster ausgelegtes Programmdesign, wie es etwa bei Nutzung von MPI [5] nötig ist, ist hier nicht mehr zwingend.

Die vorgestellte Implementierung hat jedoch noch einige Schwachstellen, die ausgebessert werden sollten.

Wenn auf ein nicht existierendes Objekt zugegriffen wird, wird dies im Augenblick noch nicht korrekt behandelt. Auch wenn Objekte – etwa durch Hardwareversagen – wirklich verloren gehen sollte, so muss der Objektmanager damit zurechtkommen können, und aussagekräftige Fehlermeldungen zurückliefern. Damit ließe sich der Schaden an dem Gesamtsystem in Grenzen halten, wenn eine Komponente ausfällt.

Eine andere Schwachstelle in der vorhandenen Implementation ist, dass die Verwaltungsstrukturen von allen Objekten immer aufgehoben werden. Um Speicherplatz zu sparen, sollten diese gelöscht werden, wenn ein Knoten lange nicht mehr auf das Betreffende Objekt zugegriffen hat und dessen Verwaltungsstruktur nicht mehr gebraucht wird.

Auch das Finden der Objekte kann noch verbessert werden. Wenn etwa ein Prozess auf einen anderen Knoten migriert und dort eine seiner Speicherseiten benötigt, muss der Knoten erst herausfinden, wo sich diese befindet. Befand sich diese Speicherseite noch nie auf dem Knoten, weiß dieser in der bestehenden Implementation nichts von deren Aufenthaltsort. Die Wahrscheinlichkeit, dass sich die Speicherseite auf dem Knoten befindet, auf dem der Prozess vorher lief, ist jedoch sehr hoch. Es muss noch untersucht werden, inwiefern es möglich ist, auf solche Abhängigkeiten zwischen einzelnen Objekten einzugehen.

Ein großes Problem in verteilten Systemen allgemein ist die Fehlertoleranz. Je mehr Komponenten beteiligt sind, um so größer ist die Wahrscheinlichkeit, dass eine ausfällt. Sehr interessant wäre es, in den Objektmanager Fehlertoleranz einzubauen. Er müsste dann Kopien der Objekte nicht nur zum Lesen anfertigen, sondern wirklich alle Objekte auf mehreren Knoten replizieren. Auch muss das Kommunikationsmodul hierfür geeignet umgebaut werden, um mit dem Ausfall eines ganzen Knotens zurechtkommen zu können. Um die Konsistenz zwischen allen Objekten zu erhalten, wäre noch viel mehr Mühe nötig, vor allem um Prozesse mit den von ihnen bearbeiteten Objekten zu synchronisieren. Dies alles ist wohl auch nicht ohne Einbussen in der Geschwindigkeit des Systems möglich. Dies ist ein sehr umfangreiches Thema, bei dem noch viele Probleme zu untersuchen sind.

Aber auch an der Effizienz des Systems kann noch gearbeitet werden. Die Nachrichtenschicht verschickt im Augenblick das nächste Paket erst, wenn das letzte bestätigt wurde. Besonders in Hochgeschwindigsnetzen könnte man in der Zwischenzeit jedoch schon weitere Nachrichten schicken, um die Hardware optimal auszunutzen. Dieses Verfahren wird auch im Internet bei Protokollen wie TCP oder RDP [10] verwendet. Hier werden häufig auch viele Meldungen auf einen Schlag bestätigt, um die Latenz so gering wie möglich zu halten. Besonders RDP ist wegen seines Nachrichtenbasieren Typs sehr interessant für dieses System.

Auch die Broadcasts könnten unter Umständen effizienter gehandhabt werden, indem diese nicht mehr für jeden Knoten einzeln gesendet werden. Hier bietet sich die Multicast-Fähigkeit moderner Netzwerkkarten an. Es gibt einige Ansätze, gesicherte Kommunikation auch bei Multicast anzubieten. Jedoch hat noch kein Protokoll den Durchbruch geschafft. Eine effiziente Lösung hierfür würde nicht nur dem vorgestellte System, sondern auch vielen anderen Projekten helfen.

Literatur

[1] **Andrew S. Tannenbaum** *Verteilte Betriebssysteme,* Prentice-Hall-Verlag, **1995**

[2] **Muskesh Singhal, Niranjan G. Shivaratri** *Advanced concepts in operating systems: distributed, databade, and multiprocessor operating systems,* McGraw-Hill, Inc., **1994**

[3] **K. Li, P. Hudak** *Memory Coherence in Shared Virtual Memory Systems,* ACM Transactions on Computer Systems, vol. 7, no. 4, **Nov. 1989**

[4] **R. E. Kessler, M. Livny** *An Analysis of Distributed Shared Memory Algorithms,* Proceedings of the 9th International Conference on Distributed Computing Systems, **1989**

[5] **W. Gropp, E. Lust, A. Skellum** *Using MPI,* MIT Press, Cambridge, MA, **1994**

[6] **L. Lamport** *Time, Clocks, and the Ordering of Events in a Distributed System,* Communications of the ACM, vol.21, no. 7, **Juli 1978**

[7] **Tim Felser** *Entwurf und Implementierung von Strategien zur Prozessverwaltung in einem verteilten Linux System,* Studienarbeit, Friedrich Alexander Universität, **2000**

[8] **Barak A., La'adan O. and Shiloh A.** *Scalable Cluster Computing with MOSIX for LINUX,* Proc. Linux Expo '99, Seiten 95-100, Raleigh, N.C., **May 1999**.

[9] **Jon Postel** *Internet Protocol,* Request For Comment 791, **1981**

[10] **David Velten, Robert Hinden, Jack Sax** *Reliable Data Protocol,* Request For Comment 908, **1984**

[11] **M. Stumm, S. Zhou Fault Tolerant Distributed Shared Memory Algorithms,** Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing, Dallas, **Dezember 1990**

[12] **K. L. Wu, W. K. Fuchs** *Recoverable Distributed Shared Virtual Memory,* IEEE Transactions on Computers, vol 39, no. 4, **April 1990**