Diplomarbeit

zum Thema

Checkpointing als Basis für transparente Service-Migration unter Linux

Jan Glauber

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

22. Januar 2002

Inhaltsverzeichnis

1 Einleitung **2**
- 1.1 Gliederung der Arbeit 3
- 1.2 Erklärung 3

2 Analyse **4**
- 2.1 Begriffsklärung 4
- 2.2 Checkpointing und Prozeßmigration 5
- 2.3 Anwendungsgebiete der Service-Migration 6
- 2.4 Anforderungen und Designentscheidungen 7
- 2.5 Prozeßzustand 9
- 2.6 Grundlegende Ansätze 12
  - 2.6.1 Implementationsebenen 12
  - 2.6.2 Restabhängigkeiten 14
  - 2.6.3 Single System Image 14
- 2.7 Phasen der Prozeßmigration 15
  - 2.7.1 Extraktion 15
  - 2.7.2 Übertragung 16
  - 2.7.3 Wiederstart 16
- 2.8 Verwandte Arbeiten 17
  - 2.8.1 Mosix 17
  - 2.8.2 Sprite 18
  - 2.8.3 SSIC 18
  - 2.8.4 MPVM 18
  - 2.8.5 Computing Communities 19
- 2.9 Checkpointing-Systeme für Linux 19
  - 2.9.1 Libckpt 20
  - 2.9.2 Condor 21
  - 2.9.3 Epckpt 21
  - 2.9.4 Crak 22
- 2.10 Zusammenfassung 24

3 Design **27**
3.1 Prozeßmigration mit Crak . . . 27
3.2 Das Unlink-Problem . . . 27
3.3 Threads . . . 29
3.4 Das PID-Problem . . . 29
3.5 Das Dateikonsistenz-Problem . . . 32
3.6 Sockets . . . 33

4 Implementation **36**
4.1 Prozeßmigration mit Crak . . . 36
4.2 Das Unlink-Problem . . . 36
4.3 Threads . . . 37
4.4 Portierung auf Linux für zSeries . . . 38
4.5 Wartezeiten . . . 39
4.6 Sockets . . . 39
4.7 Unterstützung für Mehrprozessorsysteme . . . 40
4.8 Fehler in Crak . . . 41

5 Anwendung **42**

6 Performance **45**

7 Zusammenfassung und Ausblick **47**

A Portierung von Crak für Intel i386 auf IBM zSeries **49**

B Glossar **50**

Abbildungsverzeichnis

2.1 Szenario entfernte Ausführung 5
2.2 Checkpointing Szenario 1 5
2.3 Checkpointing Szenario 2 6
2.4 Szenario Prozeßmigration 6
2.5 Taskstruktur von Linux (nach [9]) 10
2.6 Typischer Aufbau des Adreßraums eines Linux-Prozesses 11
2.7 Beispiel für eine Prozeßgruppe 11
2.8 Kommunizierende Prozesse (nach [1]) 12
2.9 Mögliche Implementationsebenen 12
2.10 Entfernte Systemaufrufe 14
2.11 Single System Image 15
2.12 Erzeugen eines Sicherungspunktes mit Crak 23

3.1 PID-Intervall Lösung 30
3.2 Dateikonsistenz zwischen Sicherungspunkten 32
3.3 Verwendung virtueller IP-Adressen 34

4.1 Gemeinsam genutzter Speicher 37
4.2 Typische Socketverbindung 40
4.3 Zustellung des STOP-Signales 41

5.1 Verwendung des GFS-Dateisystems 43

A.1 Systemaufruftest auf zSeries 49

Tabellenverzeichnis

2.1 Auswirkungen der Implementierungsebene (nach [1]) 13
2.2 Vergleich von Checkpointing-Systemen unter Linux 25
2.3 Vergleich des Codeumfangs von Migrationssystemen 25

6.1 Performance von Crak 45
6.2 NFS-Einfluß auf die Performance von Crak 46

A.1 Portierung von Crak auf Linux für zSeries 49

Kapitel 1

Einleitung

Das Internet hat die Computerwelt nachhaltig verändert. Durch die immer weiter fortschreitende Vernetzung von Computern werden neue Anwendungen denkbar. Über das Internet wird bereits eine breite Palette an Diensten für den Anwender realisiert. Neben kostenlosen Services, wie Email-Anbietern und Informationsdiensten, gibt es immer mehr Dienste, die kommerziell orientiert sind. Diese Dienste müssen anderen Ansprüchen gerecht werden, als für den Anwender kostenlose Anbieter. Es werden hohe Anforderungen an die Services gestellt, da sie weltweit verfügbar sind und viele tausende Benutzer gleichzeitig und rund um die Uhr darauf zugreifen können. Ausfälle eines Systems, die bei kostenlosen Serviceanbietern nicht kritisch sind, können hier nicht toleriert werden.

Realisiert werden die Services meist durch Cluster von Rechnern, die lokal vernetzt und nach außen über das Internet erreichbar sind. Service-Migration bezeichnet das Verschieben eines Services auf einen anderen Rechner, während der Service benutzt wird. Für Kunden, die diesen Service benutzen, soll die Migration nicht sichtbar sein. Wenn eine Service-Migration möglich ist, erhöht das die Verfügbarkeit, da es dadurch möglich wird, Rechner z. B. für Systemwartungsaufgaben aus dem Cluster zu entfernen. Den Service benutzende Kunden können dann ohne Ausfall des Dienstes weiterarbeiten. Das Ziel dieser Arbeit ist die Realisierung eines Migrationssystems, das eine vorausgeplante und transparente Migration eines Services ermöglicht.

Mit einer größeren Anzahl von Rechnern, die in einem Cluster vorhanden sind, steigt der administrative Aufwand. Die zSeries als Weiterentwicklung der klassischen IBM-Mainframes kann ein ganzes Cluster in einer Maschine vereinigen und stellt deshalb einen optimalen Ersatz für ein Cluster aus einzelnen Rechnern dar. Durch Virtualisierung bietet sie die Vorteile der unabhängigen Rechner eines Clusters, erreicht aber, aufgrund der Realisierung durch nur eine physische Maschine, eine bessere Performance. Der Trend bei Clustern geht deshalb wieder zurück zu Mainframes, daher ist diese Arbeit auch für die zSeries-Plattform interessant und erreichte Ergebnisse sollen auch dort realisiert werden.

Prozeßmigration hat sich bis heute nicht auf breiter Basis durchsetzen können. Es entstanden viele akademische Systeme, die aber keine Verbreitung gefunden haben. In erster Linie wird Prozeßmigration heute benutzt, um Lastverteilung in Clustern zu erreichen. Diese Systeme ermöglichen aber keine vollständige Migration eines Prozesses, durch die ein Service komplett auf einen anderen Rechner bewegt werden könnte. Um Prozesse vollständig zu migrieren ist es notwendig, den kompletten Zustand dieser Prozesse zu übertragen. Diese Arbeit untersucht, welche Ressourcen eines Prozesses übertragen werden müssen und welche Probleme dabei auftreten. Es wird gezeigt, daß es möglich ist, Prozesse ohne massive Änderungen des Betriebssystems zu migrieren und es werden Lösungen für Probleme bei der Prozeßmigration erarbeitet.

1.1 Gliederung der Arbeit

Im ersten Teil des zweiten Kapitels wird ein Überblick über den Stand der Technik auf dem Gebiet der Prozeßmigration gegeben. Danach werden im zweiten Kapitel für Linux vorhandene Checkpointing-Systeme analysiert. Anschließend wird näher auf ein Checkpointing-System - Crak - eingegangen. In Kapitel 3 werden mögliche Erweiterungen von Crak beschrieben. Kapitel 4 geht auf die Implementation der Erweiterungen und die Portierung von Crak auf die IBM zSeries-Architektur ein. Ein Anwendungsbeispiel und eine Performancemessung für Crak erfolgt in Kapitel 5. Im letzten Kapitel werden die Ergebnisse dieser Arbeit zusammengefaßt und ein Ausblick auf weitere, mögliche Verbesserungen erfolgt.

1.2 Erklärung

Hiermit erkläre ich, daß ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kapitel 2

Analyse

Dieses Kapitel befaßt sich, nach einer Erläuterung relevanter Begriffe, mit den Grundlagen der Prozeßmigration. Es wird die Verbindung zu Checkpointing-Systemen hergestellt und die für diese Arbeit interessantesten Systeme werden näher beschrieben. Im letzten Abschnitt erfolgt ein Vergleich der betrachteten Checkpointing-Systeme.

2.1 Begriffsklärung

Prozeß: ein Grundkonzept in allen Betriebssystemen, im wesentlichen ein Programm in Ausführung, dem ein Adreßraum zugeordnet ist [1]

> **Prozeßzustand:** alles was einen Prozeß auszeichnet; beinhaltet den kompletten Adreßraum, offene Dateien und Verbindungen mit anderen Prozessen und ist stark abhängig vom Betriebssystem

Service: ein Programm, bzw. eine Applikation, die durch einen Prozeß oder eine Gruppe von Prozessen realisiert wird und einen Dienst bereitstellt

Prozeßmigration: die Bewegung von Prozessen von einem Rechner zu einem anderen während ihrer Ausführung [8]

> **Ausgangsrechner:** der Rechner, auf dem der Prozeß bis zur Migration ausgeführt wird
> **Zielrechner:** der Rechner, auf dem der Prozeß nach der Migration ausgeführt wird

Checkpointing: das Sichern und Wiederstarten eines Prozeßzustandes, auch als *Checkpoint and Restart* bekannt

> **Checkpoint:** die durch das Speichern des Prozeßzustandes erzeugten Daten
> **Restart:** das Wiederstarten eines Prozesses aus einem vorher erzeugten Checkpoint
> **Sicherungspunkt:** wird analog zu Checkpoint verwendet

Entfernte Ausführung: *Remote Procedure Call* (RPC), der Aufruf einer Prozedur auf einem anderen Rechner

Cluster: eine beliebige Anzahl von Rechnern, die miteinander über ein Netzwerk kommunizieren können; die Rechner im Cluster vertrauen sich untereinander und es ist bekannt, welche Rechner zum Cluster gehören

2.2 Checkpointing und Prozeßmigration

Entfernte Ausführung und Checkpointing können als Abschwächungen der Prozeßmigration angesehen werden [1]. Der Unterschied zwischen den Mechanismen besteht im Umfang des betroffenen Prozeßzustandes sowie in der Art und Weise, wie der Programmcode auf dem Zielrechner ausgeführt wird.

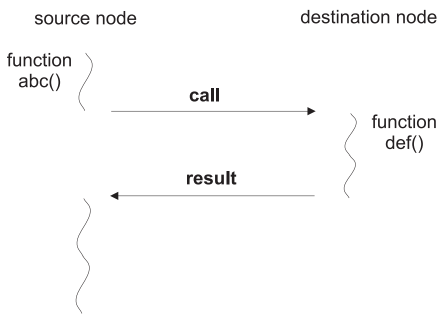

Abbildung 2.1: Szenario entfernte Ausführung

Bei der **entfernten Ausführung** wird keine Zustandsinformation über den Ausgangsprozeß übertragen, sondern nur eine Funktion mit ihren Parametern angegeben, die auf dem Zielrechner ausgeführt wird (siehe Abb. 2.1).

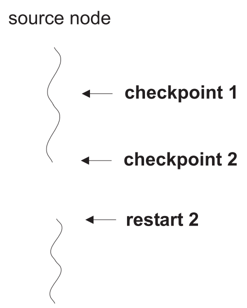

Abbildung 2.2: Checkpointing Szenario 1

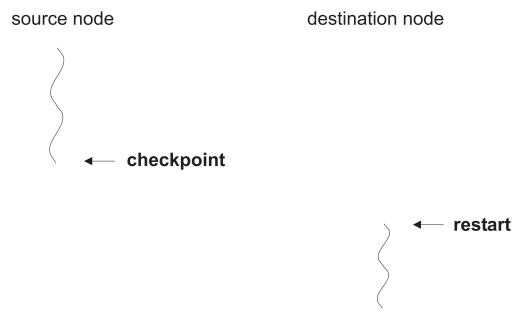

Abbildung 2.3: Checkpointing Szenario 2

Checkpointing-Systeme können nicht nur Programmcode auf einem anderen Rechner ausführen, sondern potentiell den gesamten Prozeßzustand übertragen. Traditionell werden Checkpointing-Systeme verwendet, um spezielle Anwendungen, die sehr lange laufen (z. B. mathematische Berechnungen), an beliebigen Stellen unterbrechen zu können (Abb. 2.2).

Bei Verwendung von Checkpointing ist die Ausführung auf einem anderen als dem ursprünglichen Rechner zwar möglich, aber im Gegensatz zur Prozeßmigration, bei der das Verschieben eines Prozesses auf einen anderen Rechner im Vordergrund steht, kein Entwurfsziel (Abb. 2.3).

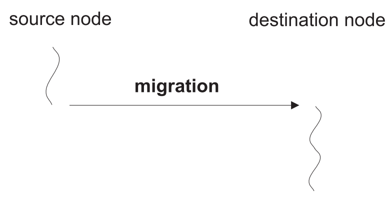

Abbildung 2.4: Szenario Prozeßmigration

Prozeßmigration ist darauf ausgerichtet, Prozesse zwischen Rechnern zu bewegen. Da potentiell alle Prozesse migrierbar sein sollen, können keine Annahmen über den Prozeß getroffen werden, d. h. der komplette Prozeßzustand muß übertragen werden. Durch ein Checkpointing-System, das den kompletten Prozeßzustand speichert, kann eine Prozeßmigration erreicht werden.

2.3 Anwendungsgebiete der Service-Migration

Dynamische Lastverteilung Basierend auf der Migration von Anwendungen ist es möglich, ein Lastverteilungssystem aufzubauen. Ist ein Rechner überlastet, so können Anwendungen, die dort laufen zu anderen, weniger ausgelasteten Rechnern im Cluster migriert werden. Dadurch kann eine gleichmäßige Verteilung von Anforderungen innerhalb einer Gruppe von Rechnern realisiert werden.

Hochverfügbarkeitssysteme Wenn es möglich ist, das Fehlverhalten eines Rechners zu diagnostizieren, können auf diesem Rechner laufende und vom aufgetretenen Fehler unabhängige Programme zu anderen Rechnern migriert werden, womit Fehler einzelner Rechner isoliert werden können.

Wird ein Checkpointing-System benutzt, das regelmäßig Sicherungspunkte eines Programmes erzeugt, diese persistent abspeichert und ein Zurücksetzen des Programmes auf den Stand eines Sicherungspunktes unterstützt, dann kann auch mit ungeplanten Ausfällen von Anwendungen umgegangen werden.

Systemwartung Ein weiteres Einsatzgebiet der Migration ergibt sich, wenn es möglich ist, alle auf einem Rechner laufenden Anwendungen zu einem anderen Rechner zu migrieren, so daß dieser Rechner aus dem Cluster entfernt werden kann. Damit wird es in einem Cluster möglich, ohne Unterbrechungen des Services einzelne Rechner z. B. für Systemwartung auszugliedern. Dadurch werden „geplante Ausfälle" eines Systems bei kontinuierlicher Verfügbarkeit eines Services möglich, wodurch sich auch die Verfügbarkeit des Systems verbessert.

Gemeinsame Ressourcennutzung Denkbar ist, daß in einem heterogenen Cluster, welches dedizierte Rechner, z. B. mit sehr viel Arbeitsspeicher enthält, Programme mit speziellen Betriebsmittelanforderungen zu einem geeigneten Rechner migriert werden.

2.4 Anforderungen und Designentscheidungen

Beim Entwurf eines Migrationssystems sind mehrere Faktoren, die teilweise im Konflikt zueinander stehen, zu beachten. Die Faktoren werden hinsichtlich ihrer Bedeutung für diese Arbeit beurteilt:

Transparenz Einem Service bzw. dem Benutzer eines Services soll die Migration einer Anwendung weitestgehend verborgen bleiben. Dies soll auch während des Migrationsvorganges selbst gelten. Nur wenn für den Benutzer eines Dienstes durch die Migration des Service kein Abbruch der Verbindung entsteht, ist Prozeßmigration sinnvoll. Es gilt, größtmögliche Transparenz anzustreben, toleriert werden muß aber die zeitliche Verzögerung durch die Migration selbst.

Skalierbarkeit und Performance Migration ermöglicht eine bessere Ausnutzung von verteilten Ressourcen, verschlechtert aber selbst die Performance durch den Mehraufwand den die Bewegung von Prozessen verursacht. Die Leistung des Systems hängt also entscheidend von der Qualität der Algorithmen ab, die bestimmen, wann und wohin ein Prozeß migriert werden soll. Ein System, welches von Migration Gebrauch macht, soll nicht langsamer sein, als das entsprechende System ohne Migration. Ebenso soll sich die Leistungsfähigkeit des Systems durch Hinzufügen von weiteren Rechnern steigern.

Vollständigkeit der Migration Viele Migrationssysteme wie z. B. Mosix [3] und Sprite [8] vereinfachen die Migration, indem ein Teil des Prozesses auf dem Rechner, wo der Prozeß ursprünglich gestartet wurde, verbleibt. Über einen Kommunikationskanal sind diese *Restabhängigkeiten* mit den migrierten Prozessen verbunden. Systemaufrufe können dadurch an den Ausgangsrechner weitergeleitet werden, wo sie ausgeführt werden. Die Ergebnisse werden zurückübertragen (siehe Abschnitt 2.6.2).
Nachteile dieser Lösung sind die Erhöhung der Fehleranfälligkeit und Performanceverluste durch die zusätzliche Kommunikation. Weiterhin kann mit teilweiser Migration nur eine Lastverteilung erreicht werden, da die verbleibenden Teile eines Prozesses ein Ausgliedern dieses Rechners aus dem Cluster verhindern.
Dieses Problem kann zwar teilweise umgangen werden, indem auf einem Rechner, der ausgegliedert werden soll, keine neuen Prozesse mehr zugelassen werden und solange gewartet wird, bis der letzte Prozeß auf diesem Rechner terminiert („schleichende“ Migration), was bei langdauernden Verbindungen nicht praktikabel ist.

Diese Arbeit untersucht Systeme, die eine *vorausgeplante, vollständige* Migration ermöglichen. Die vollständige Migration von Prozessen mit Netzwerkverbindungen ist nicht Gegenstand dieser Arbeit. Als Alternative zu einer vollständigen Migration soll nach Möglichkeit eine einfache Lösung für Applikationen, die mögliche Fehler in der Netzwerkkommunikation tolerieren können, untersucht werden.

Komplexität der Implementation Der Migrationsmechanismus kann auf verschiedenen Ebenen implementiert werden, z. B. auf Kern- oder auf Nutzerebene als Bibliothek. Der Ort der Implementation wirkt sich auf die Komplexität aus. Migrationssysteme auf Nutzerebene führen meist zu einer einfacheren Implementation [1]. Eine Realisierung auf Nutzerebene hat aber möglicherweise keinen Zugriff auf alle, für die Migration benötigten Daten und kann so keine maximale Transparenz erreichen. Eine ausführlichere Beschreibung erfolgt in Abschnitt 2.6.1.

Fehlertoleranz Minimal soll das System Fehler, die in einem Rechner auftreten, isolieren können. Dabei dürfen vom Fehler unabhängige Rechner nicht beeinträchtigt werden. Der Ausfall der Verbindung zu einem Rechner, auf dem ein Fehlerfall eingetreten ist, soll die mit ihm kommunizierenden Rechner so wenig wie möglich beeinflussen.
Vorstellbar ist auch ein System, in dem Prozesse nach Auftreten eines Fehlers auf einem Rechner zu einem anderem Rechner migriert werden. Voraussetzung dafür ist, daß der auftretende Fehler den Rechner nur teilweise beeinträchtigt, d. h. die zu migrierenden Prozesse sind nicht betroffen und das der Fehler vom System erkannt wird.

Checkpointing-Systeme sind in der Lage, im Fehlerfall auf den letzten Sicherungspunkt zurückzufallen. Sofern die Checkpoint-Daten noch verfügbar sind (z. B. auf einem zentralen Checkpoint-Server [13]), könnte der Checkpoint benutzt werden, um auf einem anderen als dem ausgefallenen Rechner den Service wiederzustarten.

Weiterhin sollte bei der Realisierung eines Migrationssystems darauf geachtet werden, das keine zentrale Instanz im Cluster vorhanden ist, von der andere Rechner abhängig sind (*Single Point of Failure*), um die Fehleranfälligkeit nicht zu erhöhen.

Heterogenität Ein erschwerender Faktor für die Prozeßmigration ist die Anwendung in einem heterogenen Cluster. Probleme treten hierbei hardware- und softwareseitig auf. Es muß beachtet werden, daß ein Prozeß ein Gerät benutzen kann, das für die Ausführung des Prozesses erforderlich und vielleicht nur auf dem Ausgangsrechner vorhanden ist. Hat der Prozeß auf dem Zielrechner keinen Zugriff mehr auf das Gerät, so kann keine Migration stattfinden.
Die Betriebssystemversionen zwischen Ausgangs- und Zielrechner können sich unterscheiden. Weiterhin können unterschiedliche Versionen einer Bibliothek zwischen den Rechnern vorhanden sein. Oder es werden gar unterschiedliche Betriebssysteme verwendet. In diesem Fall muß sichergestellt werden, das migrierte Prozesse trotz nicht-identischer Umgebungen einwandfrei weiterarbeiten können.
Probleme durch Heterogenität werden in dieser Arbeit nicht betrachtet, es wird von einem homogenen Cluster mit identischer Hard- und Software ausgegangen.

2.5 Prozeßzustand

Es sollen beliebige Prozesse, die sich einer Migration nicht bewußt sind, migriert werden können. Dazu muß der gesamte Zustand des laufenden Programmes übertragen werden, um sicherzustellen, daß das Programm ohne Einschränkungen auf dem Zielrechner ausgeführt werden kann. Dieser Zustand umfaßt neben dem Programmcode alle Ressourcen, die vom Programm verwendet werden.
Unter Linux sind diese Ressourcen in der *Taskstruktur*, die einen Prozeß beschreibt, zusammengefaßt (Abb. 2.5). Über sie sind alle Ressourcen, die für die Migration eines Prozesses benötigt werden, erreichbar.

Ressourcen, die übertragen werden müssen, beinhalten:

- alle Register, die ein Prozeß verwendet, inklusive der Spezialregister wie z. B. Gleitkommaregister,
- der komplette Adreßraum mit Code, Daten, Stack, sowie Bibliotheken (siehe Abb. 2.6); quantitativ ist der Adreßraum der größte Teil des Prozeßzustands,
- Information über alle offene Dateien, inklusive Zugriffsrechten, Dateiposition und Flags,
- der Prozeßidentifikator (PID),

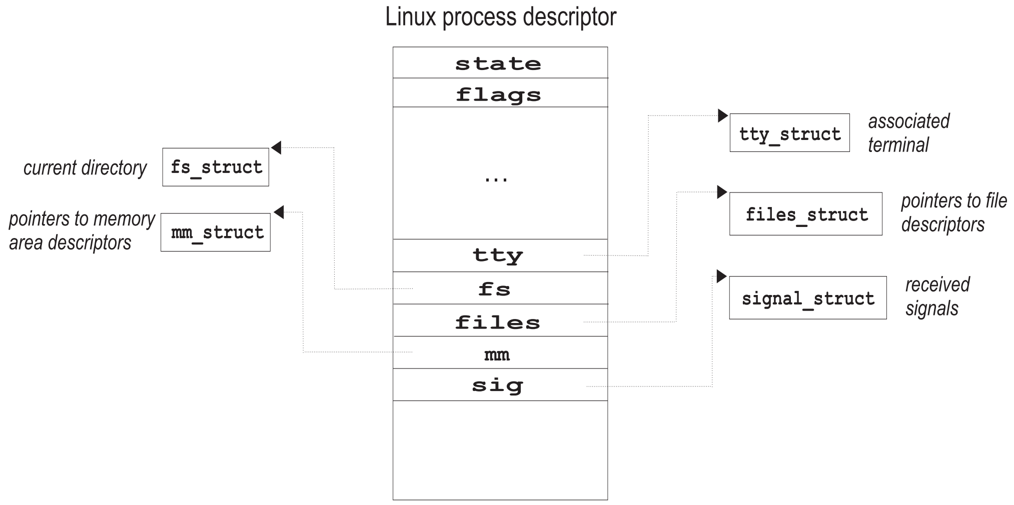

Abbildung 2.5: Taskstruktur von Linux (nach [9])

- vom Prozeß abhängige Prozesse, wenn es sich um eine *Prozeßgruppe* handelt; eine Prozeßgruppe wird durch einen Prozeß und alle seine Nachkommen gebildet (siehe Abb. 2.7),
- Information über Kommunikation mit anderen Prozessen wie *Netzwerksockets*, *IPC*, *Pipes*,
- verschiedene andere Informationen (z. B. weitere Identifikatoren, CPU-Zeiten).

Wird, wie in Abschnitt 2.7.3 beschrieben, zum Wiederstart des Prozesses auf dem Zielrechner ein neuer Prozeß erzeugt und dessen Zustand durch den zu migrierenden Prozeß ersetzt, so muß nicht die komplette Taskstruktur übertragen werden. Durch den Start eines neuen Prozesses werden alle Datenstrukturen, die ein Prozeß verwendet, korrekt initialisiert. Prozesse sind z. B. über verschiedene Listen miteinander verkettet, diese Zeiger sind in der Taskstruktur enthalten. Eine Migration dieser Daten ist nicht sinnvoll.

Problematisch sind Prozesse, die **direkt auf I/O-Geräte** zugreifen. Wenn ein Prozeß den Zustand eines Gerätes über dessen Steuerregister manipuliert und das Gerät nur lokal von einem Rechner ansprechbar ist, so kann dieser Zustand nicht berücksichtigt werden. Prozesse, die direkt Gerätespeicher manipulieren, können unter Umständen nicht migriert werden, da dieser Speicher oft nur schreibbar ist, d. h. sein Zustand kann nicht ausgelesen werden. Abhängig von der Hardware ist dieses Verhalten für eine Migration kritisch oder kann ignoriert werden. Wenn z. B. ein Ausgabegerät verwendet wird, kann eine Migration ohne Komplikationen möglich sein, da der Verlust eines Teils der Ausgabe unter Umständen nicht kritisch ist.

Ein weiteres Problem tritt auf, wenn ein Prozeß sich gerade im Kern **innerhalb eines Systemaufrufes** befindet. Durch die Migration wird der Systemaufruf abgebrochen. Beim Wiederstart würde der

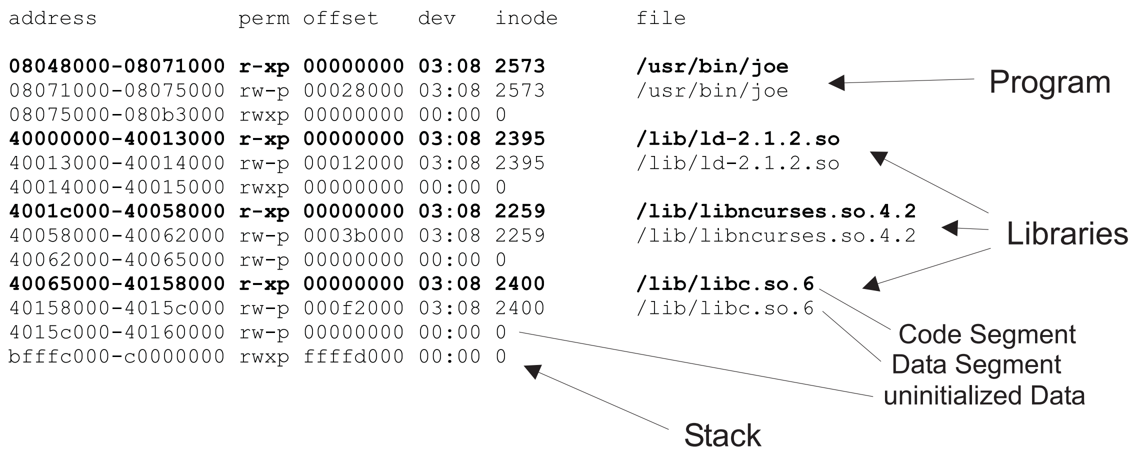

Abbildung 2.6: Typischer Aufbau des Adreßraums eines Linux-Prozesses

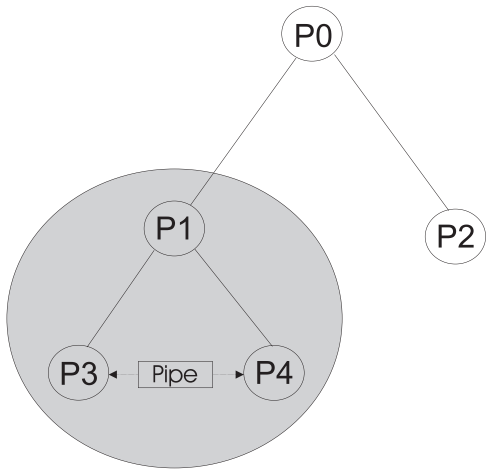

Abbildung 2.7: Beispiel für eine Prozeßgruppe

Prozeß die Ausführung mit dem Befehl nach dem Systemaufruf fortsetzen, was zu unvorhersagbarem Verhalten führen kann.

Zu beachten ist bei der Migration außerdem die **Kommunikation von Prozessen**. Wenn ein Prozeß, der migriert werden soll, mit anderen Prozessen kommuniziert (Abb. 2.8), muß sichergestellt werden, daß diese Verbindungen während der Migration nicht verlorengehen und beim Wiederstarten weiterhin vorhanden sind. Problematisch ist dies, wenn kein Zugriff auf den Zielrechner vorhanden ist. Dann ist es nicht möglich, auf dem entfernten Rechner Zustand zu sichern und Prozesse dort zu manipulieren.

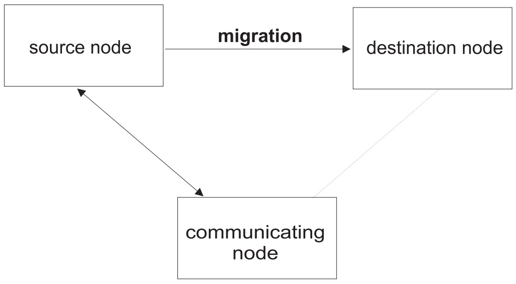

Abbildung 2.8: Kommunizierende Prozesse (nach [1])

2.6 Grundlegende Ansätze

In diesem Abschnitt werden grundlegende Möglichkeiten für die Realisierung von Prozeßmigrationssystemen diskutiert. Migrationssysteme werden abhängig von der Systemschicht, auf der sie implementiert sind, in verschiedene Klassen aufgeteilt.

2.6.1 Implementationsebenen

Grundlegend unterscheiden sich alle Migrationssysteme dadurch, auf welcher Systemebene sie aufsetzen, welche Teile des Betriebssystems sie benutzen und auf welcher Art von Betriebssystem sie aufbauen. Möglich ist die direkte Einbindung des Migrationssystems in die Applikation, indem der Programmierer explizit Funktionen zur Migration aufruft. Alternativ kann auch eine Bibliothek bereitgestellt werden, die beim *Linken* der Applikation vor das eigentliche Programm gestellt wird. Eine weitere Möglichkeit ist die direkte Einbindung des Migrationssystems in das Betriebssystem (siehe Abb. 2.9).

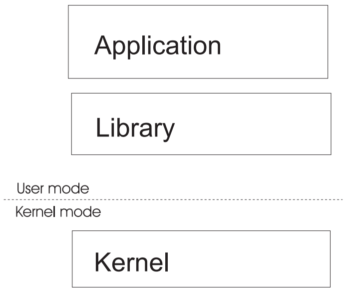

Abbildung 2.9: Mögliche Implementationsebenen

Die Implementationsebene wirkt sich auf die erreichbare Transparenz aus. Auf Kernebene kann potentiell die größte Transparenz erreicht werden, da der komplette Zustand des Prozesses erreichbar und manipulierbar ist, der Zugriff von Nutzerebene aus ist dagegen nur eingeschränkt möglich. Implementationen auf Nutzerebene sind dafür leichter portierbar und weniger aufwendig als Kernänderungen. Eine Gegenüberstellung dieser Faktoren und der Einfluß der Implementationsebene auf sie wird in Tabelle 2.1 gezeigt.

	Implementationsebene	
Faktoren	Kern	Nutzerebene
Transparenz	total	eingeschränkt
Portabilität	schlecht	sehr gut
Komplexität	hoch	gering

Tabelle 2.1: Auswirkungen der Implementierungsebene (nach [1])

Ein wichtiger Aspekt bei der Realisierung eines Migrationssystems ist die Umgebung, auf die das System aufbaut. Neben der schon beschriebenen grundlegenden Einteilung in Kern- und Nutzerebene gibt es dafür folgende Möglichkeiten:

- Erweiterung traditioneller, monolithischer Systeme (z. B. Sprite, Mosix),
- Botschaften-basierte Systeme, Mikrokerne (z. B. Mach),
- Systeme auf Nutzerebene (z. B. Condor, LSF),
- Applikationsspezifische Systeme (z. B. PVM),
- Mobile Objekte, Mobile Agenten (z. B. IBM Aglets).

Prozeßmigration in **herkömmlichen Betriebssystemen** wie Unix zu realisieren, ist am aufwendigsten, da diese Systeme sehr komplex sind und (bis jetzt) keine Unterstützung für verteilte Systeme enthalten.

Botschaften- und Mikrokernsysteme sind mit ihren Kommunikationsprimitiven schon eher geeignet, da alle Nachrichten zwischen Prozessen auf diese Primitive abgebildet werden und es leicht möglich ist, diesen Nachrichtenfluß zu manipulieren.

Migrationssysteme, die auf **Nutzerebene** (meist als Bibliothek) implementiert sind, können eventuell nicht den kompletten Prozeßzustand retten und sind damit nur für Anwendungen, die keinen von Nutzerebene unerreichbaren Zustand enthalten, interessant (siehe Abschnitt 2.7.1)

Systeme, welche die Anpassung von Anwendungen erfordern und **Mobile Objekte**, die in einer eigenen Laufzeitumgebung ausgeführt werden, sind im Rahmen dieser Arbeit nicht von Interesse.

Diese Arbeit befaßt sich mit Prozeßmigration als Erweiterung des Linux Betriebssystems. Linux zählt zu den traditionellen, monolithischen Systemen.

2.6.2 Restabhängigkeiten

Eine Methode, durch die Prozeßmigration drastisch vereinfacht wird, ist die nicht vollständige Migration. Nicht vollständig bedeutet, daß ein Teil des Prozesses auf dem Ausgangsrechner erhalten bleibt. Prozesse werden in einen Kern- und Nutzerteil aufgeteilt. Prozesse rufen über die Schnittstelle der Systemaufrufe im Kern Funktionen auf. Der Kernteil eines Prozesses besteht also aus durch den Prozeß aufgerufenen Systemaufrufen, der Nutzerteil aus den restlichen Aktivitäten des Prozesses.
Der Nutzerteil des Prozesses wird in Form seines Adreßraums und der Ausführung auf einem anderen Rechner migriert. Auf dem Ausgangsrechner bleibt jedoch ein Rest des Prozesses zurück. Zwischen Ausgangs- und Zielrechner wird ein Kommunikationskanal etabliert. Jetzt können Systemaufrufe an den Ausgangsrechner weitergeleitet werden (Abb. 2.10). Der Vorteil dieser Vorgehensweise besteht darin, daß nicht alle Ressourcen eines Prozesses migriert werden müssen. Dadurch können viele potentielle Probleme der Prozeßmigration umgangen werden (siehe Abschnitt 2.5).

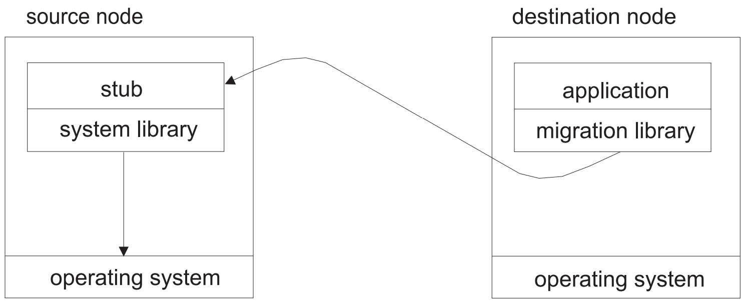

Abbildung 2.10: Entfernte Systemaufrufe

Diese Vereinfachung bedeutet aber auch, daß keine vollständige Migration mehr möglich ist, da immer ein Teil des Prozesses auf dem Ausgangsrechner verbleibt. Die Fehleranfälligkeit steigt durch Restabhängigkeiten ebenfalls, da sowohl der Ausfall des migrierten Prozesses als auch der Ausfall des Prozeßrestes oder der Verbindung zwischen beiden zu einem Fehlverhalten des Prozesses führen können. Prozeßmigrationssysteme, die Restabhängigkeiten verwenden sind Mosix [3] und Sprite [8].

2.6.3 Single System Image

Die Idee von *Single System Image* (SSI) ist es, alle Ressourcen in einem Cluster unter einem einheitlichen Namensraum zu adressieren.

Zu den Ressourcen, auf die uniform zugegriffen werden muß, zählen neben einem gemeinsamen Dateisystem auch [17]:

- IP-Adresse und Portnummer,
- Geräte,

- Speicher und
- Prozesse und Job-Management.

Durch eine uniforme Namensgebung im Cluster wird Prozeßmigration stark vereinfacht, da dadurch die größten Probleme (siehe Abschnitt 2.5) beseitigt werden und größtmögliche Transparenz in Bezug auf Ressourcen erreicht wird.

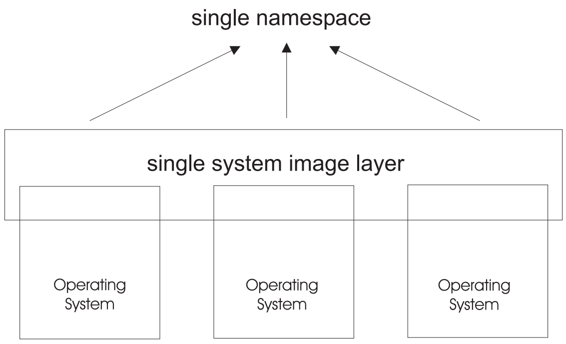

Abbildung 2.11: Single System Image

Realisiert wird SSI für herkömmliche Betriebssysteme durch eine zusätzliche Schicht, die auf dem Betriebssystem aufbaut und durch Bereitstellung eines Kommunikationsmechanismuses innerhalb des Clusters auf nicht lokal vorhandene Ressourcen zugegriffen kann (Abb. 2.11).

Linux enthält keine Unterstützung für den einheitlichen Zugriff auf Ressourcen, die vom Kern verwaltet werden und im ganzen Cluster zur Verfügung stehen sollen. Um Linux um SSI-Fähigkeiten zu erweitern, ist daher eine Änderung des Kernes notwendig.
Ein Beispiel für SSI-Systeme unter Linux ist Compaq's SSIC-Projekt [14].

Der Nachteil der SSI-Lösung liegt in den erforderlichen massiven Kernänderungen. Ein Vergleich von SSI mit anderen Lösungen erfolgt in Tabelle 2.3 S. 27. Weiterhin stellt SSI nur eine Verlagerung der Komplexität aus dem Migrationssystem in die SSI-Infrastruktur dar. Frühere Mosix-Versionen haben einen SSI-Ansatz benutzt. Dieser wurde, aufgrund der erforderlichen umfangreichen Kernänderungen, die notwendig waren, um ein SSI zu erhalten, aufgegeben [1].

2.7 Phasen der Prozeßmigration

2.7.1 Extraktion

Bevor der Zustand eines Prozesses gesichert werden kann, muß dieser angehalten werden, damit sich der Zustand nicht während des Speicherns ändert. Dafür können unter Unix Signale verwendet werden. Das Signal SIGSTOP bewirkt das sofortige Anhalten eines Prozesses[1] und kann von der Anwendung nicht blockiert werden. Wird eine Prozeßgruppe migriert, so muß jeder Prozeß der Gruppe vor der Zustandssicherung angehalten werden.

Wie einfach das Auslesen des Zustandes ist, hängt stark von der Implementationsebene ab. Arbeitet das Migrationssystem auf Kernebene, so kann potentiell der gesamte Zustand ausgelesen werden. Auf Nutzerebene ist der Zugriff eingeschränkt. Es existieren jedoch Mechanismen, um auf die prozeßeigenen Daten zugreifen zu können. Unter Unix kann z. B. ein *Signalhandler* benutzt werden, um die Register und den Adreßraum zu sichern.
Schwieriger gestaltet sich hingegen das Auslesen von Informationen über offene Dateien, da diese Attribute nicht auf Nutzerebene verfügbar sind. Probleme bereiten ebenfalls Pipes und Sockets, weil diese Kernspeicher verwenden, auf den von Nutzerebene ebenfalls nicht zugegriffen werden darf.

Nachdem der Zustand eines Prozesses ausgelesen wurde, kann der Prozeß auf dem Ausgangsrechner beendet werden. Dabei müssen alle verwendeten Ressourcen wieder freigegeben werden. Unter Linux geschieht dies automatisch bei Beenden eines Prozesses.

2.7.2 Übertragung

Die Übertragung des Prozeßzustandes kann im einfachsten Fall über ein gemeinsames Dateisystem stattfinden, welches bei vielen Migrationssystemen vorausgesetzt wird. Alternativ kann ein Prozeßzustand auch über eine Netzwerkverbindung verschickt werden.

2.7.3 Wiederstart

Der Wiederstart eines migrierten Prozesses erfolgt i. allg., indem ein neuer Prozeß erzeugt wird, der dann sukzessive durch den gespeicherten Prozeßzustand ersetzt wird. Der neue Prozeß ist während der Restaurierung gestoppt. Es wird ein Mechanismus auf dem Zielrechner benötigt, der das Ersetzen des Zustandes durchführt. Das kann entweder durch ein Programm oder durch direkten Start eines Sicherungspunktes in einem ausführbaren Format geschehen.
Bei Prozeßgruppen ist zu beachten, daß erst alle Prozesse wiederhergestellt sein müssen, bevor ein synchronisierter Neustart erfolgen kann.
Alle Ressourcen, die ein Prozeß auf dem Ausgangsrechner verwendet hat, müssen auf dem Zielrechner verfügbar und wie im Ausgangsrechner adressiert sein.

Der **Adreßraum** stellt aufgrund der virtuellen Adressierung kein Problem dar, d. h. es können auf dem Zielrechner beliebige physische Seiten benutzt werden, deren physische Adressen nicht mit den ursprünglichen Adressen übereinstimmen müssen. Eine Ausnahme stellt der direkte Speicherzugriff auf Geräte dar (siehe Abschnitt 2.5).

[1] mit Ausnahme ununterbrechbarer Systemaufrufe

Dateien müssen wieder mit ihrem ursprünglichen *Dateideskriptor* geöffnet werden und dürfen nach dem Sichern des Prozeßzustandes nicht verändert werden (siehe Abschnitt 3.2).

Die **PID** ist nicht wiederherstellbar, es gibt unter Linux keinen Mechanismus, beim Erzeugen eines Prozesses eine bestimmte PID zu verlangen (siehe Abschnitt 3.4).

Zu den **Zeiten**, die einem Prozeß zugeordnet sind, zählen *CPU-Benutzungszeiten* und *Stoppuhren*. Weiterhin kann ein Prozeß die aktuelle *Systemzeit* abfragen. Wird ein Prozeß migriert, so kann es vorkommen, daß die Systemzeiten zweier Rechner voneinander abweichen und der Prozeß eine nicht-kontinuierliche Zeit „sieht".
Benutzungszeiten eines Prozesses sind nur lokal von Bedeutung und müssen nicht berücksichtigt werden.
Stoppuhren können nicht trivial wiederhergestellt werden (siehe Abschnitt 4.5).

2.8 Verwandte Arbeiten

In diesem Abschnitt wird ein kurzer Überblick über relevante Systeme auf dem Gebiet der Prozeßmigration gegeben. Beschrieben werden nur Migrationssysteme, die traditionelle Betriebssysteme erweitern. Checkpointing-Systeme werden im nächsten Abschnitt beschrieben.

2.8.1 Mosix

Das bekannteste Prozeßmigrationssystem ist **Mosix** [3], von der Universität Jerusalem. Es existieren mittlerweile sieben verschiedene Versionen von Mosix, für unterschiedliche Hardware und Betriebssysteme. Aktuell liegt Mosix für Linux in der Version 1.0 unter der *GPL-Lizenz* vor.

Mosix wird hauptsächlich zur Lastverteilung in Clustern benutzt. Die grundlegende Technik der Restabhängigkeiten, auf der Mosix basiert, wurde in Abschnitt 2.6.2 beschrieben.

Mosix unterteilt Systemaufrufe von migrierten Prozessen in lokal ausführbare (ortsunabhängig) und in ortsabhängige Aufrufe. Ein Beispiel für Systemaufrufe, die nicht an den Ausgangsrechner übertragen werden müssen sind Aufrufe, die das Dateisystem betreffen. Da ein gemeinsames Dateisystem eingesetzt wird, sind Dateien von jedem Rechner im Cluster erreichbar. Wenn möglich, werden Systemaufrufe lokal ausgeführt, da dabei die für eine Weiterleitung erforderliche Kommunikation mit dem Ausgangsrechner entfallen kann.
Ortsabhängige Aufrufe, die nicht lokal ausgeführt werden können, sind zum Beispiel Aufrufe, welche die Netzwerkkommunikation eines Prozesses betreffen.

Die einzelnen Rechner in einem Mosix-Cluster agieren autonom, d. h. es gibt keine zentrale Instanz im Cluster. Dadurch erreicht Mosix eine gute Skalierbarkeit und verbessert die Fehlertoleranz.

2.8.2 Sprite

Sprite [8] wurde an der Universität Berkeley entwickelt. Es verwendet einen Unix-ähnlichen Betriebssystemkern, der an BSD Version 4.3 angelehnt ist. Sprite erweitert ein Unix-System zu einem verteilten Betriebssystem, in dem die einzelnen Betriebssystemkerne zusammenarbeiten. Die Kommunikation zwischen den Kernen erfolgt durch RPC. Grundlage des Systems ist ein zu Sprite gehörendes gemeinsames Dateisystem. Dies vereinfacht die Prozeßmigration, indem jede Kommunikation zwischen Prozessen auf sogenannte *Pseudo-Devices* abgebildet wird, die durch das Dateisystem realisiert werden. Dadurch kann Sprite auch Prozesse, die Netzwerkkommunikation verwenden, migrieren und erreicht so einen hohen Grad an Transparenz.

Der Server des Sprite-Dateisystems besitzt alle Informationen über offene Dateien und stellt diese bei Migration zur Verfügung, so daß offene Dateien leicht wiedergeöffnet werden können.

Sprite verwendet sowohl Restabhängigkeiten (siehe Abschnitt 2.6.2) als auch ein SSI-System (Abschnitt 2.11). Über den RPC-Mechanismus, den die einzelnen Kerne zur Kommunikation untereinander verwenden, werden Systemaufrufe, die nicht lokal ausführbar sind, an den Ausgangsrechner weitergeleitet.

Sprite realisiert ein SSI-System durch die Verwendung der Pseudo-Devices und eines einheitlichen PID-Raumes, indem PID´s um einen Identifikator für ihren Ausgangsrechner erweitert werden.

Im Gegensatz zu Mosix ist Sprite ein reines Forschungsprojekt geblieben und hat keine Verbreitung gefunden, die Arbeit an Sprite wurde 1994 eingestellt.

2.8.3 SSIC

Noch in Entwicklung befindet sich das Open-Source Projekt **SSIC** [14] von Compaq. Ausgangspunkt für SSIC ist das kommerzielle Compaq-Produkt „NonStop Clusters for Unixware“. Basierend auf einer Erweiterung von Linux um eine Cluster-Infrastruktur [18], stellt das SSIC-Projekt eine komplette SSI-Lösung (siehe Abschnitt 2.11) für Linux dar. Es beinhaltet Mechanismen zur Lastbalancierung durch Prozeßmigration. Ziele des Projektes sind Hochverfügbarkeit, Skalierbarkeit und Performance.

2.8.4 MPVM

PVM [33] ist eine Programmierumgebung für parallele, verteilte Applikationen unter Unix. Heterogene Rechner werden durch die PVM-Umgebung wie ein einzelner Rechner mit gemeinsam genutzem Speicher dargestellt, was die Erstellung verteilter Anwendungen stark vereinfacht. **MPVM** [27] ist eine Erweiterung von PVM um einen Mechanismus zur Prozeßmigration. Die Migration ist auf Nutzerebene implementiert und benutzt einen Signalmechanismus, um Prozeßzustand zu sichern. Systemaufrufe von Applikationen werden abgefangen und relevante Informationen aufgezeichnet. Dadurch kann eine Migration ohne Veränderung der Applikation erfolgen.

2.8.5 Computing Communities

Im Rahmen des Projektes **Computing Communities** [34] wurde eine vollständige, transparente Prozeßmigration unter Windows 2000 implementiert. Transparenz wird erreicht, indem alle Ressourcen eines Prozesses virtualisiert werden. Systemaufrufe einer Applikation werden an eine veränderte Bibliothek umgeleitet, die Parameter dieser Aufrufe aufzeichnet. Es ist keine Modifikation an den zu migrierenden Anwendungen erforderlich. Die Migration eines Prozesses wird in zwei Stufen durchgeführt, die Checkpoint und Wiederstart entsprechen.

2.9 Checkpointing-Systeme für Linux

Traditionell wird Checkpointing benutzt, um Sicherungspunkte während der Ausführung eines Programmes zu erzeugen. Diese beinhalten genug Zustandsinformation, um ein Programm zu einem beliebigen Zeitpunkt (auch nach Unterbrechung, zum Beispiel durch Neustarten des Rechners) an genau diesem Ausführungspunkt wiederherstellen zu können. Grundlage dafür ist, daß der Sicherungspunkt, der zum Wiederstarten verwendet wird, persistent gespeichert wird. Damit kann Checkpointing benutzt werden, um regelmäßig Rückfallpunkte eines Programmes zu speichern, wenn das im Abschnitt 3.2 beschriebene Problem der Dateikonsistenz beachtet wird.

Für Linux existieren verschiedene Checkpointing-Systeme. Grundlegend kann man diese unterteilen in Systeme, die:

- im Kern- oder Nutzermodus implementiert sind,
- die Benutzung einer Bibliothek erfordern,
- eine Anpassung des zu manipulierenden Programmes erfordern.

Checkpointing-Systeme, die im Nutzermodus implementiert sind, unterliegen dabei den schon in Abschnitt 2.6.1 angesprochenen Einschränkungen. Ohne Unterstützung vom Betriebssystem sind nicht alle verwendeten Ressourcen (bzw. deren Zustand) bekannt. So benutzen in Linux z. B. Pipes einen Puffer im Kern, der von Nutzerebene aus nicht lesbar ist. Auch Sockets und Dateien besitzen einen Zustand, der nur vom Kern aus lesbar ist. Weiterhin muß beim Wiederstart eines Prozesses, der solche Ressourcen verwendet, dieser Zustand im Kern verändert werden, d. h. es ist Schreibzugriff für Kernspeicher erforderlich. Das widerspricht aber den eingeschränkten Rechten von Programmen, die auf Nutzerebene laufen.

Alternativen, wie der Zugriff auf Informationen aus dem Kern von Nutzerebene aus erfolgen kann, werden in [15] besprochen. Dazu zählen:

- die `/proc`-Schnittstelle, über die verschiedene Informationen über Prozesse von der Nutzerebene aus gelesen werden kann,

- der ptrace-Mechanismus, der zum Debuggen entwickelt wurde und den Zugriff auf Speicher im Kern ermöglicht,
- die LD_LIBRARY_PRELOAD-Technik, durch die Bibliotheksaufrufe abgefangen und dadurch mitgelesen und verändert werden können.

2.9.1 Libckpt

Libckpt [31] ist eine Bibliothek, die Checkpointing unter Unix unterstützt. Die Bibliothek ist portabel, da sie auf Nutzerebene implementiert ist und keine Kernänderungen erfordert. Es ist erforderlich, daß Anwendungen, die diese Bibliothek nutzen wollen, angepaßt und neu übersetzt werden.
Die Bibliothek kann so konfiguriert werden, daß regelmäßig nach einem Zeitintervall (im Minutenbereich) automatisch ein Checkpoint erzeugt wird. Die Libckpt ist von den untersuchten Systemen das, mit dem geringsten Funktionsumfang. An Prozeßzustand können nur der Adreßraum, Prozeßregister und offene Dateien wiederhergestellt werden. Eingesetzt wird die Bibliothek hauptsächlich, um langandauernde Berechnungen an beliebigen Stellen unterbrechen zu können.

Libckpt konzentriert sich auf Möglichkeiten zur Performance-Optimierung des Checkpointings. Dazu werden verschiedene Techniken verwendet:

Inkrementelle Sicherungspunkte Neben dem Sichern des kompletten Prozeßzustandes besteht die Möglichkeit, durch Speichern von Checkpoints nach einem Zeitintervall im aktuellen Checkpoint nur die Änderungen zum vorhergehenden Sicherungspunkt zu speichern. Aus allen Sicherungspunkten kann dann der letzte Checkpoint zusammengesetzt werden. Das verkleinert die jeweilig zu sichernde Datenmenge.

Checkpointing unter Benutzung von `fork()` Der Systemaufruf `fork()` kopiert einen Prozeß (insbesondere seinen Adreßraum) in einen neuen Prozeß. Dieser Mechanismus eignet sich, um eine komplette Kopie eines Prozesses zu erzeugen, wobei der kopierte Prozeß weiterlaufen kann, während die Daten im erzeugten Prozeß für den Checkpoint gespeichert werden. Besonders effizient ist diese Vorgehensweise, wenn `fork()` die Copy-on-Write Methode implementiert, bei der Daten erst kopiert werden, wenn einer der beiden Prozesse diese verändert.

Benutzer-gesteuertes Checkpointing Eine weitere Optimierungsmöglichkeit wurde geschaffen, indem dem Programmierer Direktiven zur Verfügung gestellt wurden, die sich explizit auf die Sicherungspunkte auswirken. So existiert eine Anweisung, mit der Programmvariablen als „nicht im Checkpoint benötigt“ deklariert werden können.[2]
Weiterhin ist es dem Programmierer selbst möglich, an einer geeigneten Stelle einen Checkpoint durch Aufruf der Funktion `checkpoint_here()` aus der Libckpt-Bibliothek zu erzeugen.

[2] Das ist hilfreich, wenn Datenstrukturen (z. B. ein großes Array) benutzt werden, deren Inhalt zur Zeit der Erstellung des Checkpoints uninteressant ist, da er später überschrieben wird.

Um nach dem Wiederstart eines Sicherungspunktes auf vorher offene Dateien zugreifen zu können, überwacht Libckpt alle Systemaufrufe, die Dateien öffnen, und zeichnet deren Parameter auf (Dateiname, Rechte, etc.). Das ist notwendig, weil viele diese Parameter von Nutzerebene nicht erreichbar sind. Die aufgezeichneten Daten werden verwendet, um Dateien bei Wiederstart zu öffnen. Es wird davon ausgegangen, daß alle verwendeten Dateien verfügbar sind.

2.9.2 Condor

Condor [6] ist ein Lastverteilungssystem, von dem Versionen für Unix, Solaris und Linux existieren und das Checkpointing als Grundlage für Prozeßmigration benutzt. Applikationen wird dabei die Möglichkeit gegeben, sich selbst zu checkpointen. Das geschieht, indem eine Checkpointing-Bibliothek benutzt wird. Programme, die unter Condor laufen, müssen deshalb neu „gelinkt“ werden, um diese Bibliothek zu benutzen. Beim Linken wird die Standard-C-Bibliothek durch eine modifizierte Version dieser Bibliothek ersetzt.

Condor ist komplett auf Nutzerebene realisiert und erfordert keine Änderungen am Unix-Kern. Ein Checkpoint entsteht, indem die relevante Information über den Prozeß in eine Datei oder einen Socket geschrieben wird, mit dem der Prozeß dann wieder gestartet wird. Die Condor Bibliothek benutzt den Signalmechanismus von Unix, um von einem Prozeß einen Checkpoint zu erzeugen.

Neben dem Zugriff auf Dateien über ein Netzwerkdateisystem unterstützt Condor auch einen entfernten Dateizugriff ohne gemeinsames Dateisystem. Dies geschieht, indem Zugriffe auf solch einen Dateideskriptor auf Systemrufebene abgefangen und mittels RPC an einen Prozeß auf dem Ausgangsrechner zugestellt werden, der diese dann beantwortet. Damit kann ohne Netzwerkdateisystem von unterschiedlichen Rechnern aus auf die gleichen Dateien zugegriffen werden.

Eine große Einschränkung stellt, neben dem erforderlichen Neu-Linken einer Anwendung, die fehlende Unterstützung Condor's für Prozeßgruppen oder Prozesse, die neue Programme mit `exec()` starten, dar. Auch von Prozessen, die miteinander kommunizieren (z. B. über Pipes, Sockets, etc.) kann kein Checkpoint erstellt werden. Grund dafür ist, daß Condor auf Nutzerebene realisiert ist (siehe Abschnitt 2.9).

2.9.3 Epckpt

Epckpt [4] ist eine Checkpointing-Bibliothek, deren letzte Version als Kernpatch für den Linux Version 2.4.2 realisiert wurde. Dabei wird der Linuxkern um drei neue Systemaufrufe erweitert:

- `checkpoint(int pid, int fd, int flags)`: erzeugt einen Checkpoint eines Prozesses und sendet diesen an einen Dateideskriptor,
- `restart(char *ckpt_filename)`: startet einen Prozeß aus einer Checkpoint-Datei,
- `collect_data(int pid)`: veranlaßt den Kern Informationen über diesen Prozeß aufzuzeichnen, die für das Erzeugen eines Checkpoints benötigt werden.

Das Wiederstarten eines Prozesses aus einem Sicherungspunkt erfolgt, indem die Checkpoint-Datei, die im ELF-Format vorliegt, ausgeführt wird.

Epckpt ermöglicht es, komplette Prozeßhierarchien (die mit `fork()` erzeugt wurden) inklusive Prozessen, die über Pipes, *Semaphore* oder *Shared-Memory* miteinander kommunizieren, zu checkpointen. Dabei ist zu beachten, daß die Kommunikation nur zwischen Prozessen innerhalb der Prozeßgruppe erfolgen darf (siehe Abb. 2.7).

Eine Stärke von Epckpt ist die Möglichkeit, die Checkpoint-Daten nicht in eine Datei speichern zu müssen, sondern direkt an einen Dateideskriptor senden zu können, der auch auf einen Socket verweisen kann. Dadurch können die Daten direkt über das Netzwerk an einen anderen Rechner übertragen werden. Zum Wiederstarten muß der Sicherungspunkt allerdings in einer Datei vorhanden sein. Es wird davon ausgegangen, daß alle verwendeten Dateien beim Wiederherstellen des Prozesses vorhanden sind. Das kann, wenn der Prozeß auf einem anderen Rechner wiedergestartet werden soll, durch ein gemeinsames Dateisystem erreicht werden.

Die Größe des Sicherungspunktes kann reduziert werden, indem keine *Codesegmente* (des Programmes sowie der verwendeten Bibliotheken) übertragen werden. Diese Optimierung ist möglich, da Programmcode sich (normalerweise) während der Ausführung nicht ändert.

Ein Nachteil von Epckpt ist die Notwendigkeit, dem Kern beim Start eines Prozesses mitteilen zu müssen (mit dem Systemaufruf `collect_data()`), ob von diesem Prozeß potentiell Checkpoints erzeugt werden sollen. Dadurch wird es notwendig, diese Prozesse auf eine andere Art zu starten.

2.9.4 Crak

Das Checkpointing-System Crak [5] liegt für den Linuxkern 2.4.4 und 2.2.19 vor und steht unter der GPL-Lizenz. Crak ist eine Weiterentwicklung von Epckpt. Die aktuelle Version ist als Linux-Kernmodul realisiert. Nach dem Starten des Modules kann per Kommandozeile von einem beliebigen Programm ein Checkpoint erzeugt werden (siehe Abb. 2.12), welcher später wieder gestartet werden kann.

Da im Gegensatz zu Epckpt ein Kernmodul den Checkpoint erzeugt, ist es nicht möglich, Programmcode im Linuxkern zu verändern. Kernmodule bieten eine wohldefinierte Schnittstelle, um Funktionen zum Kern hinzuzufügen, sie ermöglichen jedoch nicht die Definition neuer Systemaufrufe.

Das Nutzerprogramm, welches die Erzeugung eines Checkpoints startet, ruft stattdessen über die `/dev` - Schnittstelle die entsprechende Funktion im Kernmodul auf. Das Wiederstarten eines Checkpoints erfolgt analog.

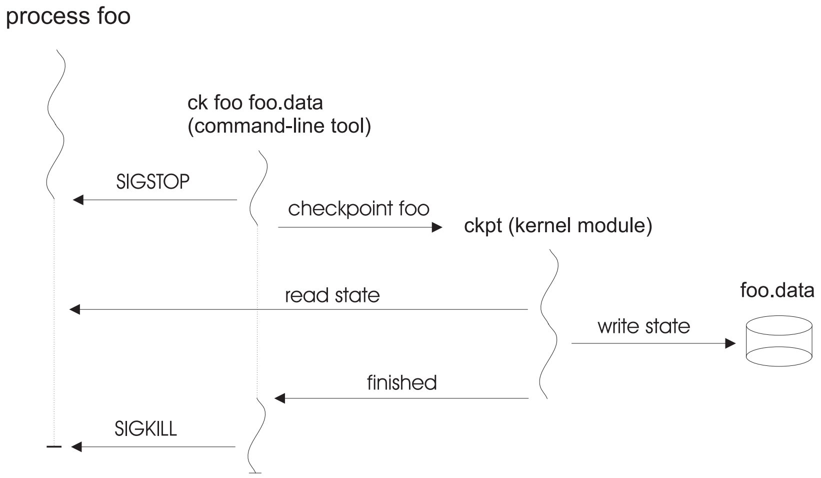

Abbildung 2.12: Erzeugen eines Sicherungspunktes mit Crak

Die Verwendung eines Modules bietet im Vergleich zu einem Kernpatch mehrere Vorteile:

- klare Trennung von Kerncode und eigenem Code,
- geringerer Entwicklungsaufwand (nach Änderungen am eigenen Code ist nicht jedesmal ein Neuübersetzen und Neustarten des Kernes erforderlich).

Der bei Epckpt nötige Aufruf zum Aufzeichnen von Systemdaten (und damit das Starten von Prozessen durch ein Dienstprogramm) entfällt. Dadurch können Anwendungen „normal“ gestartet werden, Crak kann also anhand des aktuellen Zustands eines Prozesses alle nötigen Informationen sichern. Da Crak auf Epckpt basiert, verfügt Crak im wesentlichen über die gleichen Fähigkeiten.

Implementationsdetails von Crak

In diesem Abschnitt wird beschrieben, wie Crak auf Ressourcen zugreift und sie beim Wiederherstellen des Prozesses restauriert:

Prozessorregister Es gibt eine Verbindung zwischen dem Prozeßdeskriptor und den Registern, die einem Prozeß zugeordnet sind. Diese wird benutzt, um die Register zu speichern. Register des Prozesses können beim Wiederstart vom Kernmodul aus auf ihre alten Werte zurückgesetzt werden.

Adreßraum Um auf den Adreßraum eines beliebigen Prozesses zugreifen zu können, werden die *globalen Kernmappings* verwendet. Der physische Speicher ist unter Linux ab einer bestimmten Adresse eingeblendet. Um also eine beliebige Seite eines beliebigen Adreßraumes auslesen zu können, muß eine Umrechnung der virtuellen Adresse in die physische erfolgen.
Trifft Crak auf eine *ausgelagerte Speicherseite*, so muß diese Seite zum Speichern wieder eingelagert werden. Dafür ruft Crak die entsprechende Funktion des Linuxkerns auf und speichert die

Seite anschließend. Speicherseiten werden zuerst in einen Kernpuffer kopiert und anschließend in der Checkpoint-Datei gespeichert. An dieser Stelle ist eine Optimierung möglich, indem der Inhalt der Speicherseiten direkt in die Checkpoint-Datei kopiert wird.

Bei der Wiederherstellung des Prozesses wird der Adreßraum restauriert, indem mittels `mmap()` die Daten aus dem Sicherungspunkt direkt eingeblendet werden, d. h. es findet kein Kopieren in den Speicher statt, sondern es werden direkt die Daten aus der Checkpoint-Datei benutzt. Die Seiten werden als *Copy-On-Write* markiert, was den Vorteil hat, daß der Wiederstart sehr schnell erfolgt und Seitenfehler erst verzögert beim ersten Schreibzugriff auftreten.

Dateien Dateien werden direkt durch das Wiederstart-Dienstprogramm auf Nutzerebene restauriert. Dazu werden sie entsprechend dem Original wiedergeöffnet. Dateiposition und Kopien von Dateideskriptoren (durch `dup()` erzeugt) werden restauriert.

Synchronisation Zu Beginn des Checkpointings werden alle betroffenen Prozesse angehalten[3]. Nach Erstellen des Checkpoints werden die Prozesse beendet (falls bei Aufruf des Checkpointing-Programmes nicht anders angegeben).
Wiederhergestellte Prozesse befinden sich während der Restaurierung im angehaltenen Zustand. Wird eine Prozeßgruppe wiedergestartet, so wird ein Kontrollprozeß verwendet. Dieser sendet, wenn alle Prozesse wiederhergestellt sind, ein Signal zur Fortsetzung eines angehaltenen Prozesses (SIGCONT) an alle Prozesse, wodurch diese ihre Ausführung fortsetzen.

2.10 Zusammenfassung

Crak bietet von den untersuchten Checkpointing-Systemen den größten Funktionsumfang (vgl. Tabelle 2.2). Aus diesem Grund wurde Crak als Ausgangspunkt für diese Arbeit gewählt. Die vorgestellten Checkpointing-Systeme Condor und Libckpt erfordern Änderungen am zu migrierendem Programm, weshalb sie nicht in Frage kamen. Epckpt diente als Grundlage für Crak, wurde aber in vielen Aspekten verbessert.

Der Ansatz, das Checkpointing-System auf Kernebene zu implementieren für das Erreichen maximaler Transparenz am vielversprechendsten.

Es gibt bereits eine Reihe von Systemen, die Prozeßmigration realisieren (siehe Abschnitt 2.8). Warum soll ein Checkpointing-System als Grundlage für Prozeßmigration eingesetzt werden, anstatt auf vorhandenen Prozeßmigrationssysteme aufzubauen? Der Grund dafür ist, daß existierende Prozeßmigrationssysteme auf eine Anwendung in einem Cluster zielen und dafür die Betriebssysteme, auf denen sie aufbauen meist stark erweitern (vgl. Tabelle 2.3).

[3] durch Senden des Signals SIGSTOP

	Libckpt	Condor	Epckpt	Crak
Implementation	Bibliothek	Bibliothek	Kernpatch	Kernmodul
Applikationen	neu übersetzen	neu linken	wrapper	keine Änderungen
Funktionsumfang:				
Adreßraum	ja	ja	ja	ja
Prozeßgruppen	-	-	ja	ja
offene Dateien	ja	ja	ja	ja
Signale	-	ja	-	ja
Pipes	-	-	ja	ja
IPC	-	-	ja	-
Terminals	-	-	-	ja
Sockets	-	-	-	-
Übertragung	Datei	Dateideskriptor	Dateideskriptor	Dateideskriptor

Tabelle 2.2: Vergleich von Checkpointing-Systemen unter Linux

	Mosix	Compaq SSI	Crak
modifizierte Kerndateien:	124	323	0
Quellcode Dateien:	58	59	7
Lines-of-Code:	42096	127871	2295

Tabelle 2.3: Vergleich des Codeumfangs von Migrationssystemen

Das Betriebssystem wird zu einem verteilten Betriebssystem umgebaut, bei dem die einzelnen Systeme miteinander kooperieren. Weiterhin zielen diese Systeme auf Anwendungen wie Lastbalancierung. Es werden Einschränkungen im Hinblick auf die Vollständigkeit der Prozeßmigration, die nicht Ziel, sondern nur Mittel ist, in Kauf genommen (siehe Abschnitt 2.6.2).

Das Ziel dieser Arbeit ist ein Prozeßmigrationssystem, welches minimale Änderungen am zu Grunde liegenden System erfordert und gleichzeitig eine totale Migration ermöglicht. Checkpointing-Systeme sind diesem Ziel näher als Prozeßmigrationssysteme. Deshalb erschien eine Anwendung der Checkpointing-Systeme als der geeignetere Weg, um eine transparente Migration ohne umfangreiche Systemänderungen zu erreichen.

Probleme, die von keinem der untersuchten Checkpointing-Systemen gelöst wurden:

- Dateien, die mittels `unlink()` gelöscht wurden können nicht wiederhergestellt werden,
- Prozesse können nicht unter der gleichen PID wiedergestartet werden,
- es werden keine Threads unterstützt,
- es ist nicht möglich, von Prozessen die Sockets verwenden einen Checkpoint zu erstellen.

Von Programmen, die auf diese Ressourcen angewiesen sind, kann aufgrund der fehlenden Unterstützung kein Checkpoint erstellt werden, bzw. es können Fehler nach dem Wiederstart auftreten.

Kapitel 3

Design

Aufgrund der Erkenntnisse aus Abschnitt 2.10 wurde Crak als Ausgangspunkt für diese Arbeit ausgewählt. Dies erschien anhand des Funktionsumfangs von Crak sinnvoll, um nicht von Grund auf ein neues Migrationssystem zu entwerfen. Dieses Kapitel beschreibt den Entwurf von Erweiterungen, die im Rahmen dieser Arbeit an Crak vorgenommen wurden. Es werden Lösungsansätze für die im letzten Abschnitt beschriebenen Probleme erarbeitet.

3.1 Prozeßmigration mit Crak

Crak stellt zwei Kommandozeilendienstprogramme zur Verfügung, zum Checkpointen und Wiederstarten von Programmen. Bei einer Prozeßmigration wird der gespeicherte Prozeßzustand automatisch auf einen anderen Rechner übertragen und gestartet. Um dieses Verhalten mit Crak nachzubilden, war es am einfachsten, die von Crak bereitgestellten Programme zu benutzen und mit einem Skript so zu erweitern, daß eine Prozeßmigration entsteht.

Die Übertragung des Sicherungspunktes kann dabei prinzipiell nach einem beliebigen Mechanismus - wie in Abschnitt 2.7.2 beschrieben - geschehen. Aufgrund der Voraussetzung, daß alle verwendeten Dateien in einem gemeinsamen Dateisystem verfügbar sein müssen, kann auch der Sicherungspunkt in diesem gespeichert werden.

Es muß ein Signalmechanismus implementiert werden, der nach erfolgtem Checkpointing auf dem Zielrechner den Wiederstart veranlaßt.

3.2 Das Unlink-Problem

Crak kann nicht mit Programmen umgehen, die den Systemaufruf `unlink()` benutzen. Dieses Verhalten stellt ein Problem dar, da dieser Aufruf unter Unix i. allg. dazu verwendet wird, temporäre Dateien anzulegen.

Dabei wird eine neue Datei erzeugt, worauf die Anwendung einen Dateideskriptor für diese Datei erhält. Direkt nach der Erzeugung wird der Dateisystemeintrag mit `unlink()` entfernt. Nun ist die Datei nur noch über den Dateideskriptor ansprechbar, der nur dieser Anwendung zur Verfügung steht. Wird der Deskriptor freigegeben, wird auch der belegte Platz auf dem Speichermedium freigegeben.
Wenn von einem Programm, welches mit `unlink()` eine Datei gelöscht hat und nur noch einen offenen Dateideskriptor besitzt, ein Sicherungspunkt erstellt wird, so wird der Dateiname[1] zwar im Checkpoint vermerkt, aber nach Beendigung des Programmes wird der Dateideskriptor geschlossen, womit die Datei verschwindet. Beim Versuch, diesen Checkpoint neuzustarten, tritt ein Fehler auf, weil die Datei nicht vorhanden ist.

Um dies zu verhindern, muß der Inhalt dieser Dateien gespeichert werden. Die erste Idee war, beim Checkpointen mittels `dup()` den Dateideskriptor zu duplizieren, was ein Löschen der Datei verhindert hätte. Das erwies sich allerdings als nicht sinnvoll, da damit folgende Probleme auftreten:

- das Checkpointing-Modul, durch das der Checkpoint erzeugt wurde darf, nicht beendet werden, da sonst der Dateideskriptor verloren wäre,
- beim Wiederstarten muß der Dateideskriptor an das Programm zurück übertragen werden,
- die Übertragung des Dateideskriptors muß bei Neustart auf einem anderen Rechner über Rechnergrenzen (und damit zwischen zwei Linuxkernen) erfolgen.

Aus diesen Gründen erschien die erste Idee nicht praktikabel. Eine bessere Variante, welche die oben genannten Probleme umgeht, ist es, zu versuchen, die betroffenen Daten als reguläre Datei zu speichern. Der Wiederstart wäre damit ohne Änderungen möglich, die Daten würden einfach in einer regulären Datei, unter einem anderen Namen[2] vorhanden sein. Die einzige Einschränkung ist, daß eine Datei, die normal beim Beenden der Applikation gelöscht wurden wäre, im Dateisystem „übrig“ bleibt.
Für das Speichern der Daten der aus dem Dateisystem gelöschten Datei gibt es zwei Möglichkeiten:

1. Da beim Checkpointen ein Dateideskriptor vorhanden ist, können die Daten in eine neu erzeugte Datei kopiert werden. Die neue Datei wird dann im Checkpoint anstelle der gelöschten Datei vermerkt, beim Neustart wird diese reguläre Datei benutzt.
2. Beim Checkpointen werden die Auswirkungen des `unlink()`-Aufrufes rückgängig gemacht, die Datei wird wieder zu einer regulären Datei.

Der erste Ansatz hat den Nachteil, daß eine zusätzliche Kopieroperation anfällt. Bei vermutlich großen temporären Dateien würde der Checkpointing-Vorgang damit erheblich länger dauern.
Beim zweiten Ansatz entfällt dieses Kopieren, dafür wird ein Eingriff in das Dateisystem vorgenommen, der abhängig vom verwendeten Dateisystem ist. Da die zweite Strategie performanter ist, wurde sie zur Implementation ausgewählt.

[1] NFS benennt diese Dateien um, nach z. B.: .nfs0340745903874
[2] der ursprüngliche Name ist nach dem Aufruf von `unlink()` verloren

3.3 Threads

Crak unterstützt keine Threads. Seit Version 2.2 ist im Linuxkern ein Threadkonzept enthalten. Dazu wurde der Kern um den Systemaufruf `clone()`, mit dem Threads erzeugt werden können, erweitert. Threads entsprechen unter Linux im wesentlichen Prozessen, sie erhalten einen eigenen Prozeßdeskriptor. Der Unterschied zu Prozessen besteht in der Möglichkeit, Ressourcen mit dem erzeugenden Prozeß zu teilen. Bei dem Aufruf von `clone()` kann angegeben werden, ob:

- Speicher,
- Dateien,
- Signalhandler,
- sowie Arbeits- und Stammverzeichnis

gemeinsam benutzt werden sollen.
Werden Threads beim Checkpointen wie Prozesse behandelt, bedeutet das also, daß z. B. der Adreßraum, obwohl identisch, zwischen einem Prozeß und einem von diesem erzeugten Thread doppelt gespeichert wird. Um dies zu verhindern war es notwendig, gemeinsam genutzte Ressourcen zu entdecken und nur einmal abzuspeichern.

Bei Wiederstarten des Checkpoints müssen gemeinsame Ressourcen beachtet werden. Threads müssen also entsprechend ihrem Originalzustand vor dem Checkpoint mit identischen Ressourcen wiederhergestellt werden.

3.4 Das PID-Problem

Ein Prozeßidentifikator (PID) ist ein innerhalb eines Betriebssystems eindeutiger Name, der jeweils nur einem Prozeß zugeordnet wird. Er wird verwendet, um Prozesse einfach adressieren zu können und besteht i. allg. aus einer Zahl.

Die Prozeßnummern sind nur lokal eindeutig, d. h. in einem Cluster werden PID's pro Rechner vergeben. PID's werden für folgende Systemaufrufe verwendet:

- `waitpid(pid, flags)`: wartet auf einen bestimmten Prozeß,
- `kill(pid), signal(pid)`: sendet ein Signal an einen Prozeß.

Beim Erzeugen eines neuen Prozesses bekommt der erzeugende Prozeß die PID des Kindes mitgeteilt. Ebenso kann ein Prozeß jederzeit seine eigene PID mittels `getpid()` abfragen.

Problematisch sind PID's, wenn ein Prozeß migriert wird. PID's werden unter Linux willkürlich ausgewählt, es ist nicht möglich für einen zu erzeugenden Prozeß, eine bestimmte PID anzufordern.

Deshalb haben mit Crak migrierte Prozesse nach der Migration eine andere PID als vorher. Das kann zu Problemen führen, wenn vor der Migration die PID abgefragt und gespeichert wurde. Nach dem Wiederstart ist sie nicht mehr gültig. Es ist nicht möglich, diese gespeicherten PID's zu finden, da eine PID eine 32-Bit-Zahl[3] ist und beginnend von 0 gezählt wird.

Wenn ein Prozeß nach dem Neustart eine veraltete PID benutzt, führt dies zu unvorhersagbarem Verhalten.

Mögliche Lösungen des PID-Problems:

1. Single System Image
2. PID-Intervalle
3. Ersetzen betroffener Systemaufrufe

Eine SSI-Lösung kommt für Crak nicht in Frage, da die notwendigen Änderungen dem Ansatz von Crak, keine Modifikationen am Kern vorzunehmen widersprechen (siehe Abschnitt 2.10).

Die zweite Idee scheint sehr einfach realisierbar zu sein. Innerhalb des Clusters wird das verfügbare PID-Intervall so aufgeteilt, so daß eine PID jeweils nur auf einem Rechner vergeben wird. Voraussetzung dafür ist, daß jeder Rechner im Cluster seinen eigenen PID-Raum zugewiesen bekommt. Das kann im einfachsten Fall statisch, beim Hochfahren des Rechners geschehen, z. B. durch die Initialisierungsfunktion eines automatisch geladenen Kernmoduls.

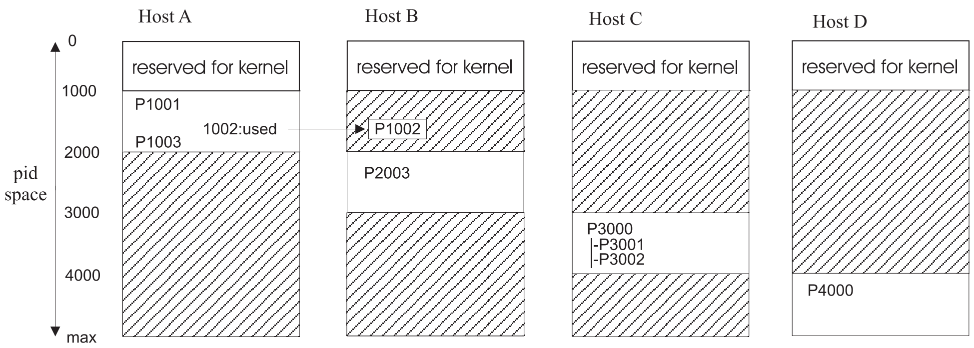

Abbildung 3.1: PID-Intervall Lösung

Der nächste Schritt ist das Sperren aller PID's, die nicht auf dem Rechner verfügbar sein sollen. Der naive Ansatz dafür wäre, solange Prozesse zu erzeugen, bis keine PID's mehr verfügbar sind und dann alle Prozesse mit PID's aus dem erlaubten Intervall zu beenden, so daß diese Prozeßnummern wieder verfügbar sind. Diese Idee ist nicht sinnvoll, da pro Prozeß ein Prozeßdeskriptor angelegt

[3]aus Kompatibilitätsgründen mit älteren Unix-Varianten werden allerdings nur 15 Bit benutzt

wird, der 8 KB groß ist. Bei 32768 möglichen Prozeßnummern ergibt dies einen Speicherbedarf von 268 MB, was nicht akzeptabel ist.

Wenn es möglich ist, PID's zu reservieren, müßten lokal gestartete Prozesse protokolliert werden, so daß deren PID's nicht mehr verwendet werden. Bei einer Migration ist die entsprechende PID immer auf dem Zielrechner verfügbar, weil sie dort gesperrt wurde. So ist ein Wiederstart unter der originalen PID möglich (Abb. 3.1).

Nachteile, der Intervall-Lösung sind also, das ein Weg gefunden werden muß, PID's zu sperren, ohne dafür einen Prozeßdeskriptor anzulegen.
Weiterhin ist der PID-Raum mit 2^{15} relativ klein. Bei einer großen Anzahl von Rechnern in einem Cluster kann das pro Rechner resultierende Intervall bei einer statischen Aufteilung zu klein werden. Probleme ergeben sich, wenn ein Prozeß, der migriert wurde, auf einem anderen Rechner als auf dem er erzeugt wurde, beendet wird. In diesem Fall wäre es notwendig, den Ausgangsrechner zu benachrichtigen, daß die PID wieder verfügbar ist, sonst vergrößert sich das Problem der zu kleinen PID-Intervalle. Dadurch wird eine zusätzliche Kommunikation erforderlich. Wenn ein Rechner im Cluster ausfällt und neu gestartet wird, so muß diesem mitgeteilt werden, welche Prozesse in seinem Intervall erzeugt und migriert wurden. Es ist notwendig, Information über migrierte Prozesse aufzuzeichnen.
Durch die erforderliche zusätzliche Kommunikation und das Aufzeichnen von Information über migrierte Prozesse wird die, auf den ersten Blick leicht umzusetzende Intervall-Lösung, aufwendiger zu implementieren.

Ein Kernpatch kann das PID-Problem durch Ersetzen der betroffenen Systemaufrufe lösen. Wenn festgestellt wird, daß ein Programm, von dem ein Checkpoint gemacht wurde seine PID abfragt, kann dem Programm die ursprüngliche PID zurückgegeben werden. Dafür muß eine Tabelle der Prozesse und ihrer PID´s gespeichert werden.

Da Crak bisher vollständig ohne direkte Kernänderungen realisiert wurde, sollte ein Kernpatch vermieden werden. Eine Alternative für eine Manipulation von Systemaufrufen (ohne diese zu ändern) ist das Abfangen der Aufrufe durch Benutzung einer veränderten Bibliothek.
Systemaufrufe können durch eine Linux Umgebungsvariable `LD_LIBRARY_PRELOAD` an eine modifizierte Bibliothek weitergeleitet werden. Damit ist eine Manipulation wie durch einen Kernpatch möglich.

Wie leicht realisierbar eine Lösung des PID-Problems durch PID-Intervalle ist, hängt entscheidend von der Größe des PID-Raums ab. Ist dieser groß genug (z. B. durch 32-Bit große PID's), so kann auf eine Wiederverwendung einmal benutzter PID's verzichtet werden. Wenn es weiterhin möglich ist, PID's ohne Platz- und Zeitverschwendung zu blockieren, wäre die Intervall-Lösung für eine Implementation die beste Wahl.

3.5 Das Dateikonsistenz-Problem

Im Gegensatz zu Transaktionssystemen, die zu jedem beliebigen Zeitpunkt auf einen vorherigen Sicherungspunkt zurückfallen können, ermöglichen Checkpointing-Systeme ein Zurückfallen bzw. einen Neustart nur unter bestimmten Bedingungen.

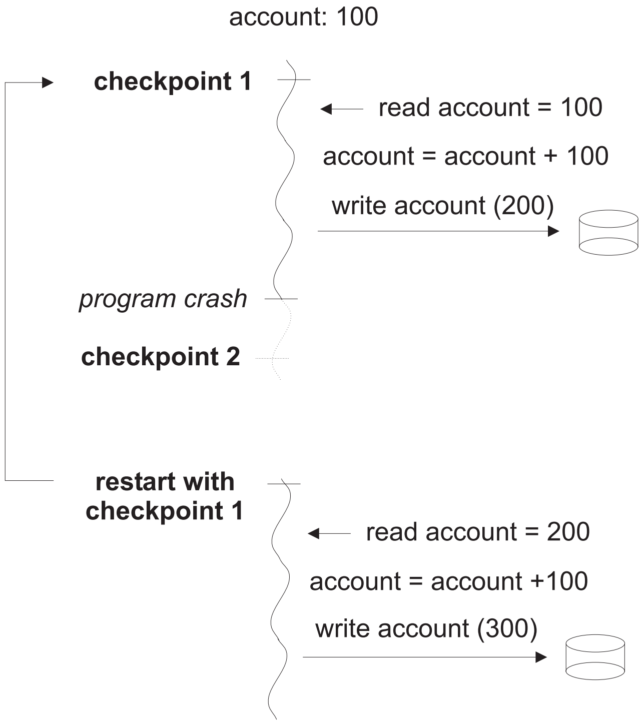

Abbildung 3.2: Dateikonsistenz zwischen Sicherungspunkten

Checkpointing-Systeme beinhalten kein Konzept um sicherzustellen, daß Dateien zwischen Checkpoint und Wiederstart nicht verändert werden. Das klassische Beispiel für einen daraus resultierenden Fehler ist in Abb. 3.2 zu sehen.

Inkonsistenzen können auf zwei verschiedenen Wegen entstehen:

1. Es ist möglich, daß ein anderer Prozeß eine Datei verändert, die das Programm, von dem ein Checkpoint erzeugt wird, ebenfalls verwendet.
2. Wird ein Programm nach einem Checkpoint nicht sofort beendet, und es wird nach Änderungen an einer Datei auf den letzten Sicherungspunkt zurückgefallen, kann dies ebenfalls zu einem nicht wiederstartbarem Zustand führen.

Das zweite Szenario kann nur bei Checkpointing-Systemen auftreten, da bei Prozeßmigration das Programm auf dem Ausgangsrechner nach der Migration beendet wird.

Für das Dateikonsistenz-Problem kommen folgende Lösungen in Frage:

1. Ignorieren oder Verbieten von Änderungen
2. Erkennen der Änderung (leicht möglich durch das Überprüfen des Zeitstempels des letzten Schreibzugriffes auf die Datei oder durch Quersummen)
3. Kopien von allen, vom Programm benutzten, Dateien zum Sicherungspunkt hinzufügen [16]
4. inkrementell alle Änderungen im Vergleich zu einer ursprünglichen Version einer Datei im Sicherungspunkt speichern

Die erste Ansatz ist sinnvoll, wenn auf die Dateien nur lesend zugegriffen wird oder die Änderungen an den Dateien nicht kritisch sind.
Der zweite Ansatz dient nur der Erkennung und kann in Verbindung mit der ersten Lösung eingesetzt werden.
Die dritte und die vierte Lösung sind sehr aufwendig zu realisieren. Beim kompletten Sichern aller Dateien wächst die Größe des Sicherungspunktes. Inkrementelle Änderungen verschlechtern die Performance durch Herausfinden und Wiederanwenden der Änderungen.

3.6 Sockets

Es ist mit Crak nicht möglich, Programme, die Sockets verwenden zu checkpointen. Zum Zeitpunkt der Erstellung dieser Arbeit war kein Migrationssystem mit Unterstützung für Socketmigration bekannt. Verschiedene Gruppen beschäftigen sich nach Eigenaussagen mit der Entwicklung von Socketmigration [3], [4], [5], [14]. Bis jetzt ist keine Implementierung, die das Problem löst, bekannt.

Im Rahmen dieser Arbeit sollte ein anderer Ansatz untersucht werden. Bei der Netzwerkkommunikation über Sockets können Fehler auftreten, deren Behandlung dem Applikationsprogrammierer überlassen ist. Die Idee ist, anstelle einer Migration eines Sockets die Verbindung abzubrechen und der Anwendung einen geeigneten Fehlercode zurückzuliefern, so daß diese, wenn sie Fehler korrekt behandelt, die Verbindung wieder aufsetzen kann. Das bedeutet, daß Anwendungen einen potentiellen Ausfall der Netzwerkverbindungen tolerieren müssen.

Weiterhin muß sichergestellt werden, daß durch den Ausfall der Verbindung keine Inkonsistenzen beim Wiederstart entstehen. Ein Socket wird unter Linux durch einen Dateideskriptor repräsentiert. Wird dieser bei Wiederstart nicht geöffnet, so kann der Deskriptor, der auf dem Ausgangsrechner auf einen Socket verwiesen hat, beim nächsten Öffnen einer Datei vergeben werden. Wenn die Anwendung dann auf ihren Socket zugreifen will, entsteht ein Programmfehler, der für die Anwendung nicht vorhersehbar ist.

Daraus folgt, daß minimal ein Socket unter dem gleichen Dateideskriptor wiedergeöffnet werden muß. Das muß mit den gleichen Parametern wie für den ursprünglichen Socket geschehen. Es ist also erforderlich, beim Erstellen des Sicherungspunktes alle zum Wiederöffnen eines Sockets erforderlichen Parameter zum Sicherungspunkt hinzuzufügen.

Parameter, die sich bei Wiederstart des Sockets nicht trivial restaurieren lassen, sind die *IP-Adresse* sowie der dazugehörige *Port*. Wird ein Prozeß auf einem anderem Rechner wiedergestartet, so ändert sich dadurch die IP-Adresse. Ein Client, der davon nichts weiß, versucht jedoch auf der alten IP-Adresse eine neue Verbindung aufzubauen. Da der Vorgang für der Client transparent sein soll und im TCP/IP-Protokoll kein Mechanismus existiert, eine Änderung einer IP-Adresse anzuzeigen, ist eine andere Herangehensweise an das Problem notwendig.

Eine Möglichkeit ist, nach außen einen Service über eine feste IP-Adresse bekannt zu machen, aber den Service im eigenen Netz auf einem beliebigen Rechner (also unter verschiedenen IP-Adressen) bereitzustellen. Lösungen für dieses Problem werden in [20], [22] und [23] beschrieben.

Ein Ansatz ist die Verwendung eines *Gateway* (siehe Abb. 3.3), der nach außen sichtbar ist und eingehende Service-Anfragen innerhalb des Clusters an den verantwortlichen Server weiterleitet. Der Client sieht dabei nur die IP-Adresse des Gateway, die sich nicht ändert. Innerhalb des Clusters können die Services auf beliebigen IP-Adressen laufen, deren IP-Adresse nur dem Gateway für die Weiterleitung der Anfragen bekannt sein muß. Die Antworten des Service werden entweder zum Gateway, oder direkt zum Client zurückgesendet.

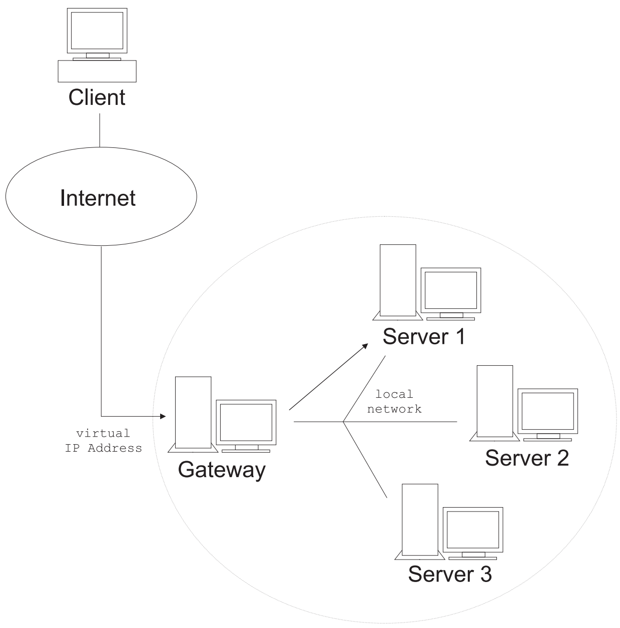

Abbildung 3.3: Verwendung virtueller IP-Adressen

Bei Wiederaufbau der Verbindung zu einem Service der migriert wurde, muß neben der IP-Adresse auch sichergestellt werden, daß der gleiche Port wie auf dem Ausgangsrechner verwendet wird. Es wird davon ausgegangen, daß der Port noch nicht belegt ist.

Kapitel 4

Implementation

In diesem Kapitel wird die Umsetzung, der im letzten Kapitel beschriebenen Erweiterungen, die im Rahmen dieser Arbeit an Crak vorgenommen wurden, beschrieben. Es wird weiterhin die Portierung auf die zSeries-Architektur erklärt und Lösungen für Probleme, die beim Einsatz von Crak auftraten, werden beschrieben.

4.1 Prozeßmigration mit Crak

Um den Checkpointing-Vorgang mit Crak zu automatisieren, wurde ein Perl-Skript implementiert, welches auf dem Zielrechner gestartet wird und auf eine Nachricht auf einem UDP-Socket wartet. Auf dem Ausgangsrechner wird ein Perl-Skript gestartet, dem die PID des zu migrierenden Prozesses übergeben wird. Das Skript ruft das Checkpointing-Dienstprogramm von Crak auf und wartet auf die Erzeugung des Sicherungspunktes. Ist die Checkpointing-Phase beendet, wird eine Nachricht an den Zielrechner gesendet, welcher daraufhin mit dem Wiederstart beginnt. Der Sicherungspunkt muß im gemeinsamen Dateisystem vorliegen. Der Dateiname des Sicherungspunktes, der erzeugt wurde, wird in der Nachricht übertragen.

4.2 Das Unlink-Problem

Die Voraussetzung um mit Dateien, deren Verzeichniseintrag gelöscht wurde, umgehen zu können ist, diese Dateien von regulären Dateien zu unterscheiden. Da die Verzeichnisinformation wiederhergestellt werden sollte (siehe Abschnitt 3.2) und Crak zusammen mit dem NFS-Dateisystem benutzt wurde ist die Implementation NFS-abhängig. Die Anpassung an ein anderes Netzwerkdateisystem würde aber vermutlich keine größeren Änderungen erfordern.

Der Systemaufruf `unlink()` bewirkt im NFS-Dateisystem, daß die Statusinformation der Datei im Dateisystem[1] geändert wird. Wird nun der letzte Dateideskriptor freigegeben, so wird die Datei

[1] das Dateiflag wird auf DCACHE_NFSFS_RENAMED gesetzt

gelöscht. Um die Datei wieder in eine reguläre Datei umzuwandeln (die Auswirkungen von `unlink()` rückgängig machen), wird:

1. das Dateiflag zurückgesetzt
2. der Verzeichniseintrag wieder hergestellt

Die Wiederherstellung kann durch einen `open()` Aufruf erfolgen, da die Datei nur umbenannt wurde. Im Kern ist kein `open()` Aufruf verfügbar, da es sich um eine Bibliotheksfunktion auf Nutzerebene handelt. Um aus dem Linuxkern heraus Systemaufrufe durchführen zu können, gibt es in Linux einen Mechanismus, bei dem durch ein Makro `_syscallX(return, call, args ...)` mit variabler Parameteranzahl jeder Systemaufruf aus dem Kern heraus angesprochen werden kann.

Normalerweise werden Dateien, auf die `unlink()` ausgeführt wurde, nach Programmende gelöscht. Wenn dies für den Wiederstart übernommen worden wäre, so würde die Datei nach dem Wiederstart des Checkpoints verschwinden. Da ein Sicherungspunkt auch mehrmals wiedergestartet werden kann, war dies nicht erwünscht. Es wurde ein Flag definiert, womit der Benutzer beim Wiederstarten des Sicherungspunktes steuern kann, ob diese Dateien nach Programmende gelöscht werden oder erhalten bleiben sollen.

4.3 Threads

Um zu erkennen, ob ein Prozeß oder ein Thread vorliegt, reicht es aus, gemeinsame Ressourcen zu erkennen (siehe Abschnitt 3.3). Die gemeinsame Benutzung von Ressourcen ist leicht festzustellen über deren Referenzzähler. Gemeinsame Ressourcen werden in Linux realisiert, indem die entsprechenden Zeiger der Taskstruktur auf die gleichen Ressourcen verweisen.

```
if ((p->mm->mm_users.counter > 1) &&
    (p->p_pptr->mm == p->mm))
     *clone_flags |= CLONE_VM;
```

Abbildung 4.1: Gemeinsam genutzter Speicher

Information über gemeinsame Ressourcen wird in der Checkpoint-Datei mit gespeichert und beim Neustart berücksichtigt. Der Neustart von mehr als einem Prozeß erfolgt, indem die Kindprozesse mit `fork()` erzeugt werden. Werden statt Prozessen Threads verwendet, so werden diese mit `clone()` und den beim Checkpointen festgestellten gemeinsamen Ressourcen wiedergestartet.

Für die Benutzung von Threads empfiehlt Linux[2] die Verwendung einer Threadbibliothek. Eine weitverbreitete Threadbibliothek, die unter Linux auch in der Systembibliothek *libc* enthalten ist,

[2] vgl. man-page zu clone()

ist die *Libpthread*. Sie ermöglicht die Verwendung POSIX-kompatibler Threads. Beim Checkpointen von Programmen, welche diese Bibliothek verwenden, können Probleme im Zusammenhang mit veränderten PID's (siehe 3.4) entstehen. Die Threadbibliothek besitzt eine Tabelle zur Umsetzung der für den Benutzer sichtbaren Threadidentifikatoren in Prozeßnummern. Dadurch sind die Threadidentifikatoren an die entsprechenden PID's gebunden und eine Veränderung der PID durch den Wiederstart eines Prozesses aus einem Checkpoint würde zu einem Fehlverhalten, bei Zugriff auf diese Tabelle führen.
In einem Checkpointing-System, daß auf Libckpt [31] basiert und Linux-Threads unterstützt [16] wurde dieses Problem gelöst, indem diese *Mappings* durch Aufruf der bibliotheksinternen Funktion `map_pid_tid()` nach Neustart aus einem Checkpoint neu erzeugt wurden. Dieses Vorgehen hat den Nachteil, das Implementationsdetails der Threadbibliothek verwendet werden und die Lösung nur mit dieser Threadbibliothek funktioniert. Wenn es gelingt, das Problem der veränderten Prozeßnummern zu lösen (vgl. Abschnitt 3.4), kann auf einen derartigen Eingriff verzichtet werden.

4.4 Portierung auf Linux für zSeries

Die IBM zSeries ist der aufwärtskompatible Nachfolger der S/390-Architektur. Die zSeries-Plattform ist eine 64-Bit Maschine mit herausragender Leistung im I/O-Bereich. Sie zeichnet sich durch einen hohen Grad an Parallelisierbarkeit aus und stellt partitionierbare, virtuelle Maschinen zur Verfügung. Diese können aufgrund der speziellen Hardware der zSeries-Architektur untereinander über sehr schnelle, virtuelle Netzwerkverbindungen miteinander kommunizieren.
In den letzten Jahren ist die Verwendung von Clustern zur Realisierung von rechenintensiven Aufgaben immer populärer geworden. Eine zSeries kann ein ganzes Cluster ersetzen, indem eine entsprechende Anzahl virtueller, voneinander unabhängiger Maschinen auf der zSeries eingerichtet wird. Vorteile sind der geringere administrative Aufwand und höhere Leistung durch die Punkt-zu-Punkt Verbindungen.

Im Gegensatz zu vielen anderen Großrechnern gibt es für die zSeries-Plattform eine vollständige Linux-Portierung. Da Linux ein Betriebssystem ist, das für verschiedene Hardwareplattformen verfügbar ist, wurde es in einen architekturunabhängigen und einen (minimalen) architekturabhängigen Teil aufgeteilt.
Crak wurde original für die Intel i386-Plattform geschrieben. Bei der Portierung von Crak auf die zSeries Architektur sind also nur Codeteile, die sich auf den architekturabhängigen Teil von Linux beziehen zu beachten.

Das trifft bei Crak zu auf:

- Systemaufrufneustart,
- Performancemessung und
- verwendete Register.

Systemaufrufneustart wenn ein Programm in einem Systemaufruf unterbrochen wird, und dieser wiederstartbar ist (der Systemaufruf wird abgebrochen und der Rückgabewert ist `EINTR`), bleibt es dem Anwendungsprogrammierer überlassen, den Systemaufruf erneut auszuführen.
Crak startet unterbrochene Systemaufrufe automatisch und transparent für den Benutzer neu. Dazu wird der Befehlszeiger zurückgesetzt, so daß der nächste ausgeführte Befehl nach Rückkehr zur Anwendung der Systemaufruf ist. Das Register `EAX` wird auf den entsprechenden Wert zurückgesetzt. Es muß beim Checkpointen festgestellt werden, ob sich das Programm in einem Systemaufruf befindet. Dazu werden auf dem i386 die Inhalte der Register `EAX` sowie `ORIG_EAX` überprüft.
Auf der zSeries sind Systemaufrufe anders implementiert, so daß dieser Mechanismus angepaßt werden mußte (siehe Abb. A.1 im Anhang).

Zur Unterstützung von **Performancemessungen** wird auf i386 das *Timestamp-Counter* Register verwendet. Auf der zSeries wurde zur Messung von Zeiten das dort entsprechende Register eingesetzt.

Andere verwendete Register ließen sich einfach auf die entsprechenden Register der zSeries abbilden (siehe Tabelle A.1 im Anhang).

4.5 Wartezeiten

Als Nebeneffekt bei einem Systemaufrufneustart können falsche Wartezeiten entstehen. Ein Prozeß kann mit dem Systemaufruf `sleep(int seconds)` ein bestimmtes Intervall warten. Wird der Prozeß in diesem Systemaufruf unterbrochen, so wird der Aufruf neu aufgesetzt und die bereits abgelaufene Wartezeit ist verloren, d. h. es wird noch einmal die komplette Wartezeit absolviert.

Weiterhin sieht ein Prozeß der migriert wurde und die Systemzeit abfragt keine kontinuierliche Zeit, da zwischen Checkpoint und Wiederstart beliebig viel Zeit vergehen kann. Da dieses Verhalten keine schwerwiegenden Auswirkungen auf Programme hat, werden diese Probleme zur Zeit ignoriert.

4.6 Sockets

Das Ziel war es, eine minimale Unterstützung von Sockets durch Crak zu implementieren. Dazu ist es zuerst notwendig zur Checkpoint-Zeit Sockets und ihre Optionen zu erkennen. Dies ist leicht möglich durch die Verbindung der Taskstruktur mit den Socketoptionen.

Bei Wiederstart aus einem Checkpoint werden Sockets auf Nutzerebene durch den Systemaufruf `socket()` wieder geöffnet. Der typische Ablauf einer Socketverbindung ist in Abb. 4.2 zu sehen. Da Crak wie in Abschnitt 4.4 beschrieben einen Systemaufruf, in dem sich der Prozeß zur Checkpoint-Zeit befindet wiederstartet, fehlen bei Wiederöffnen eines Sockets durch `socket()` die vorherigen Systemaufrufe, die diesen Socket manipuliert haben. Bei einem Serverprozeß trat beim Wiederstart dadurch ein Fehler auf, der von einer Anwendung kaum erwartet werden kann, da er durch das Checkpointing des Prozesses erzeugt wurde.

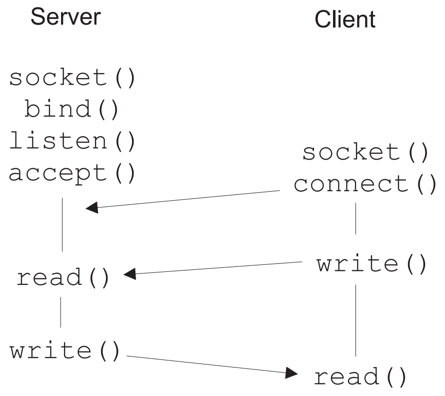

Abbildung 4.2: Typische Socketverbindung

Sockets, die mit `accept()` auf einkommende Verbindungen warten, sind an Hand ihres Zustand-Flags zu erkennen (`TCP_LISTEN`). Um Servern einen Wiederstart zu ermöglichen, wurden bei Auftreten eines solchen Sockets die fehlenden Systemaufrufe nach dem Öffnen des Sockets ausgeführt. Die erforderlichen Parameter für `bind()` und `listen()` sind in den beim Checkpointing gespeicherten Socketoptionen enthalten.

4.7 Unterstützung für Mehrprozessorsysteme

Crak ist schon für den Einsatz auf Rechnern mit mehreren Prozessoren (SMP) vorbereitet. Beim Zugriff auf Kerndaten werden bereits an allen relevanten Stellen Synchronisationsmechanismen in Form von *Spinlocks* benutzt.
Beim Einsatz von Crak auf einer zSeries mit mehreren Prozessoren traten Probleme auf. Der Checkpointing-Prozeß begann Zustand des Prozesses zu sichern, obwohl der Prozeß noch lief. Das bedeutet, daß sich Zustand des Prozesses noch ändern kann und nicht mehr mit den Daten im Sicherungspunkt übereinstimmt, was zum Fehlerfall bei Wiederstart führen kann. Das Problem lag an der Annahme, das bei Beginn des Checkpointings sich das System im Kernmodul befindet und kein weiterer Prozeß ausgeführt wird. Diese Annahme trifft aber nur auf Einprozessorsysteme zu.

Wenn das Checkpointen eines Prozesses durch Aufruf des Kommandozeilenprogramms beginnt, so wird dem Prozeß ein Stopsignal gesendet. Um sicherzustellen, daß ein Prozeß zu Beginn des Checkpoints durch das Kernmodul wirklich angehalten ist (vgl. Abschnitt 4.7), wird bei Beginn des Checkpoints dessen Zustand abgefragt und, solange der Prozeß noch läuft, gewartet (siehe Abb. 4.3).

Der Wiederherstellen von Prozeßgruppen erfolgt bereits synchronisiert, wie in Abschnitt 2.9.4 beschrieben.

```
while (process->state != TASK_STOPPED)
  schedule();
```

Abbildung 4.3: Zustellung des STOP-Signales

4.8 Fehler in Crak

Crak ist noch in der Entwicklung begriffen. Obwohl das Grundsystem zuverlässig funktioniert, treten in bestimmten Situationen (abhängig von den Programmen, von denen ein Checkpoint erstellt wird) Fehler auf.

Die Implementation der Erkennung von Prozeßgruppen ist mangelhaft und derzeit auf zwei Ebenen (d. h. ein Prozeß und ein Kindprozeß) beschränkt.

Das Dateiformat, in dem der Sicherungspunkt gespeichert wird, benötigt eine Überarbeitung. Zur Zeit enthält es für die verschiedenen Ressourcen Abschnitte, die optional sind, wenn die Ressourcen eventuell nicht vorhanden sind. Das gilt aber nicht für alle Ressourcen, die im Sicherungspunkt gespeichert werden. Ein einheitlicher Aufbau aller Sektionen mit Verweisen zwischen den Abschnitten würde die Handhabung der gespeicherten Daten erleichtern.

Es wurden folgende Fehler von Crak im Rahmen der Arbeit korrigiert:

- Dateien wurden eventuell mit falschen Deskriptoren wiedergeöffnet,
- die Dateiposition wurde beim Wiederstart nicht gesetzt,
- Speicherseiten wurden nicht immer korrekt abgespeichert,
- die Prozeß-ID der `/proc` Einträge wurde nicht korrekt gesetzt.

Kapitel 5

Anwendung

Als Fallbeispiel, an dem die Funktion Crak's demonstriert werden soll, dient der *Lock-Server* des *GFS-Dateisystems*. Dieser realisiert die für die Synchronisation des Dateizugriffes notwendigen Sperren. Die Verwaltung der Sperren erfolgt komplett im Hauptspeicher, was die Performance verbessert. Durch Checkpointing des Lockservers wird es möglich, den Lockserver auf einem anderen Rechner wiederzustarten, was geplante Ausfälle des Servers ermöglicht.

GFS ist ein verteiltes Dateisystem, das folgende Vorteile im Vergleich zu NFS bietet:

- es ist kein zentraler Server notwendig und damit kein Single Point of Failure vorhanden,
- bessere Performance.

Es ist erforderlich, Zugriffe auf das Dateisystem zu koordinieren. Bei NFS war dafür der schon für die Bereitstellung der Dateien über das Netzwerk verantwortliche Server zuständig. Bei GFS wird auf einen Server verzichtet, die Dateisysteme werden direkt von den einzelnen Rechnern angesprochen. Zum Zugriff auf nicht-lokal vorhandene Dateisysteme kann das *Network Block Device* verwendet werden, welches über eine Netzwerkverbindung den transparenten Zugriff auf entfernte Geräte gewährleistet.
Für die Koordination der Zugriffe gibt es mehrere Möglichkeiten. GFS kann über eine definierte Schnittstelle in Form von Modulen verschiedene Locking-Mechanismen benutzen. In Entwicklung befindet sich eine Erweiterung des SCSI-Protokolles, die direktes Locking auf der Platte durch Einsatz von SCSI-Kontrollbefehlen ermöglichen soll.

Auf der zSeries ist es möglich, eine Platte zwischen mehreren Instanzen der virtuellen Maschine zu teilen (siehe Abb. 5.1). Dadurch kann auf den Einsatz des Network Block Device Treibers verzichtet werden.

GFS ist derzeit in der Alpha-Version OpenGFS 4.01 unter der GPL verfügbar. Eine schon weiterentwickelte Version 4.2 wurde von der ursprünglichen Entwicklergruppe der Firma Sistina unter einer kommerziellen Lizenz veröffentlicht.

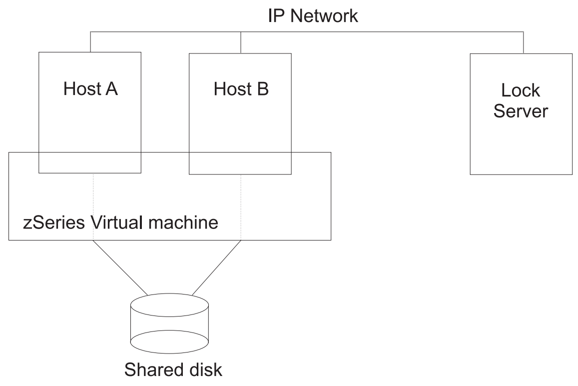

Abbildung 5.1: Verwendung des GFS-Dateisystems

Deshalb fiel die Entscheidung auf die leider noch sehr instabile, aber frei verfügbare OpenGFS-Version.

Derzeit ist für das Locking ein Server verantwortlich, was den Nachteil der erhöhten Fehleranfälligkeit hat. Fehlertoleranz kann derzeit nur erreicht werden, indem alle Locks auf Festplatte protokolliert werden, wodurch ein Wiederstart des Servers erfolgen kann. Dieses Vorgehen ist aber extrem performancelastig, da bei jeder Anforderung eines Locks auf die Platte geschrieben werden muß.
Mit Crak wäre es möglich, einen geplanten „Umzug“ des Lockservers zu einem beliebigen Zeitpunkt und unter Last durchzuführen, wenn z. B. der Ausgangsrechner heruntergefahren werden muß. Für einen Umzug des Locking-Servers ohne Crak müßte dieser wie beschrieben alle Zugriffe auf Platte speichern.

Da GFS-Client und Lockserver über Sockets kommunizieren ist die Voraussetzung für eine erfolgreiche Migration, daß eine robuste und fehlertolerante Implementation der Netzwerkzugriffe, wie in Abschnitt 3.6 beschrieben, vorliegt.

Das Ergebnis des Versuchs war, daß die erfolgreiche Migration des Lockservers gelang, was zeigt, daß die in Crak implementierte Socketunterstützung als mögliche Lösung für das Checkpointing von Prozessen, die Netzwerkverbindungen verwenden, in Frage kommt.
Die GFS-Clienten reagieren auf einen Ausfall der Verbindung, indem sie versuchen, wieder den Server zu kontaktieren. Solange dies nicht gelingt, werden Prozesse, die Dateien des GFS-Dateisystems verwenden, blockiert. Wird der Server aus dem Checkpoint wiedergestartet, so bauen die Clienten die Verbindung wieder auf und die Ausführung der angehaltenen Prozesse wird fortgeführt.

Beim Wiederstarten des Lockservers auf einem anderen Rechner ist zu beachten, daß sich die IP-Adresse ändert. Um dieses Problem zu lösen, können virtuelle IP-Adressen verwendet werden, wie in Abschnitt 3.6 beschrieben. Aus Zeitgründen wurde dies im Rahmen der Diplomarbeit nicht getestet.

Für die Arbeit mit Crak wurden, um die korrekte Arbeitsweise der vorhandenen Funktionen zu überprüfen, eine Vielzahl von Tests implementiert. Diese Programme haben jeweils einzelne Prozeßressourcen verwendet. Mit Crak wurde ein Checkpoint des Programmes erzeugt und wiedergestartet. Durch diese Tests wurden verschiedene Fehler in Crak gefunden (siehe Abschnitt 4.8).

Kapitel 6

Performance

Es wurden für diese Arbeit erste Performancemessungen mit Crak durchgeführt, die im Rahmen der erwarteten Ergebnisse (verglichen mit [5] und [4]) lagen. Detaillierte Messungen erschienen nicht sinnvoll, da an Crak noch keine Optimierungen hinsichtlich der Performance vorgenommen wurden.

Gemessen wurde die Zeit vom Beginn der Erzeugung des Checkpoints, bis zum Ende des Wiederstarts auf einem anderen Rechner. Tabelle 6.1 vergleicht die Zeiten für Checkpointing ohne Optimierung (der Programmcode der Applikation und aller Bibliotheken wird in den Checkpoint aufgenommen) und mit Optimierung.

	Checkpointing	Restart	Checkpoint Size
with binaries:	1843 ms	2 ms	1.7 MB
without binaries:	85 ms	2 ms	387 KB

Tabelle 6.1: Performance von Crak

Interpretation der Meßergebnisse Die Zeit, die benötigt wird, um einen Checkpoint zu erzeugen, hängt hauptsächlich von der Größe des vom Programm verwendeten Speichers ab. Im Vergleich zum Adreßraum eines Prozesses ist der restliche gespeicherte Zustand vernachläßigbar klein. Die identischen Zeiten für den Wiederstart eines Prozesses entstehen dadurch, daß bei der Restaurierung des Adreßraums direkt die Daten aus dem Sicherungspunkt in den Adreßraum des Prozesses „eingeblendet" werden, wie in Abschnitt 2.9.4 beschrieben.

Großen Einfluß auf die Performance hat das Dateisystem, welches zum Speichern des Sicherungspunktes verwendet wird. Für diese Arbeit wurde Crak zusammen mit dem NFS-Dateisystem benutzt. Einflüsse des NFS-Dateisystems sind in Tabelle 6.2 zu sehen. Die Zeit, die für das Checkpointen benötigt wird, hängt hauptsächlich vom Umfang des Checkpoints ab. Der Unterschied zwischen der Größe des Checkpoints und den zum Speichern benötigten Zeiten entsteht, weil bei den Messungen Caches, die das Dateisystem verwendet, nicht berücksichtigt wurden.

Die gemessenen Zeiten für eine Migration liegen im Rahmen der Anforderungen dieser Arbeit. Für einen Service, der über ein nicht echtzeitfähiges Netzwerk wie das Internet benutzt wird, ist eine Verzögerung im Sekundenbereich tolerierbar, da schon in der Netzwerkkommunikation solche Unterbrechungen auftreten können.

	Checkpointing	NFS write	Checkpoint Size
with binaries:	1843 ms	1410 ms	1.7 MB
without binaries:	85 ms	14 ms	387 KB

Tabelle 6.2: NFS-Einfluß auf die Performance von Crak

Testumgebung Als Ausgangsrechner wurde ein Pentium mit 133 Mhz und einer 100 Mbit Ethernet-Netzwerkkarte verwendet. Zielrechner war ein 100 Mhz Pentium mit einer 10 Mbit Ethernet-Netzwerkkarte. Beide Rechner enthielten 64 MB Hauptspeicher.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurde, basierend auf dem Checkpointing-System Crak, gezeigt, daß eine vollständige Prozeßmigration durch ein Checkpointing-System unter bestimmten Bedingungen möglich ist. Es wurden Lösungen für die Einschränkungen, denen Crak unterlag (vgl. Abschnitt 2.10) erarbeitet und zum größten Teil implementiert.
Crak ist damit das erste Checkpointing-System für Linux, das eine generelle Unterstützung für Prozesse, die Threads verwenden, bietet[1]. Weiterhin wurde eine Lösung für das Problem der gelöschten Dateien (vgl. Abschnitt 3.2) implementiert.
Auch Applikationen, die Fehler in den Netzwerkverbindungen abfangen, können jetzt mit Crak migriert werden. Das Konzept für eine minimale Socketunterstützung durch Crak wurde durch Checkpointing des GFS-Lockservers erfolgreich getestet. Weitere Tests mit Applikationen, die Sockets verwenden, sind erforderlich.

Crak befindet sich noch im Entwicklungszustand, d. h. es sind noch an vielen Stellen Optimierungen möglich. Besonders betrifft dies das Sichern des Adreßraums, da dies die umfangreichste Ressource darstellt. Verbessert werden kann die Art, wie Speicherseiten kopiert werden (eine statt zwei Kopieroperationen, siehe Abschnitt 2.9.4). Weiterhin speichert Crak auch nicht-initialisierte Seiten (Speicherseiten, die initialisiert, auf die aber noch nicht zugegriffen wurde) im Sicherungspunkt. Diese Seiten werden aber unter Linux alle auf eine „Nullseite“ abgebildet. Crak sollte das berücksichtigen.

Die in der aktuellen Version von Crak fehlende Unterstützung für Shared Memory, Semaphore und Message Queues (System V IPC) sollte leicht implementierbar sein, da sie in der früheren Versionen 2.2.19 von Crak schon vorhanden war und somit nur an die aktuelle Linux-Version angepaßt werden muß.
Für das Problem der Prozeßidentifikatoren wurden verschiedene mögliche Lösungen diskutiert. Anhand dieser Ideen ist zu erwarten, daß auch dieses Problem ohne Änderungen am Betriebssystem lösbar ist. Mit der erfolgreichen Implementation einer Lösung des PID-Problems wäre ein vollständi-

[1] die im Gegensatz zu [15] unabhängig von der Implementation einer Threadbibliothek ist

ges Prozeßmigrationssystem für Linux vorhanden. Basierend auf der schon realisierten Unterstützung für Sockets kann ein Konzept für die vollständige Socketmigration erarbeitet werden.

Anhang A

Portierung von Crak für Intel i386 auf IBM zSeries

Im diesem Abschnitt werden die Änderungen, die notwendig waren, um Crak von der i386- auf die zSeries-Architektur zu portieren (vgl. Abschnitt 4.4) gegenübergestellt.

	Intel i386	IBM zSeries
verwendete Register	`eax`	`gprs[2]`
	`orig_eax`	`orig_gpr2`
	`esp`	`gprs[15]`
	`eip`	`psw.addr`
	`eflags`	`psw.mask`
Performancemessung	`rdtsc`	`stck`

Tabelle A.1: Portierung von Crak auf Linux für zSeries

Für die Implementation des Systemaufrufneustarts war es erforderlich festzustellen, ob sich ein Prozeß in einem Systemaufruf befindet. Auf Intel i386 geschiet dies durch einen Vergleich des Programmcodes, an dem sich der Prozeß bei Erstellung des Checkpoints befindet. Dabei wird der Befehl, auf den der *Instruction Pointer* zeigt, mit dem Code eines Systemaufrufs verglichen.

Auf der zSeries war diese Vorgehensweise nicht möglich. Es wurde stattdessen ein Makro aus dem Linuxkern verwendet:

```
regs->trap == __LC_SVC_OLD_PSW
```

Abbildung A.1: Systemaufruftest auf zSeries

Anhang B

Glossar

Adreßraum: der virtuelle Speicher, der mit einem Prozeß assoziiert ist

Dateideskriptor: eine Zahl, der symbolisch für eine offene Datei steht

ELF: Binärformat für ausführbare Dateien

GFS: Global File System, verteiltes Dateisystem ohne zentralen Server

GPL: GNU Public Licence, populäre Open-Source Lizenz

IPC: Interprocess Communication, Kommunikation zwischen Prozessen

Kern: das Betriebssystem

Kernebene: Privilegstufe, auf der das Betriebssystem ausgeführt wird

NFS: Networking File System, ein gemeinsames Dateisystem für nur durch ein Netzwerk verbundene Computer

Nutzerebene: Privilegstufe, auf der Nutzerprogramme mit eingeschränkten Rechten ausgeführt werden

PID: eine für alle unter einem Betriebssystem laufenden Prozesse eindeutige Prozeßnummer

Pipe: eine Punkt-zu-Punkt Verbindung zwischen zwei Prozessen

Prozeß: ein Programm in Ausführung

Semaphore: Mechanismus zur Synchronisation zwischen Prozessen

Shared Memory: gemeinsamer Speicher zwischen Computern

Signale: asynchrone Nachrichten an Prozesse

Signalhandler: Funktion, die bei Eintreffen eines Signales aufgerufen wird

SMP: Symmetric Multiprocessing, Verwendung mehrerer Prozessoren in einem Rechner

Socket: bezeichnet eine Netzwerkverbindung

Spinlock: Datenstruktur, die einen synchronisierten Zugriff auf Ressourcen durch wechselseitigen Ausschluß realisiert

Taskstruktur: Struktur, mit der das Betriebssystem Prozesse beschreibt

Literaturverzeichnis

[1] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, Songnian Zhou, *Process Migration*, 1999

[2] Milojicic, Douglis, Wheeler, *Mobility - Processes, Computers and Agents*, ACM Press, 1999

[3] Amnon Barak, Oren La'adan, Amnon Shiloh, *Scalable Cluster Computing with MOSIX for LINUX*, Hebrew University of Jerusalem, 1999

[4] Eduardo Pinheiro, *Truly-Transparent Checkpointing of Parallel Applications (Working Draft)*, Federal University of Rio de Janeiro, http://www.cs.rutgers.edu/ edpin/epckpt/

[5] Hua Zhong, *Crak: Linux Checkpoint/Restart As Kernel Module*, University of Columbia, New York, 2000, http://www.cs.columbia.edu/ huaz/english/research/crak.htm

[6] Michael Litzkow, Todd Tannenbaum, Jim Basney, Miron Livny, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, University of Wisconsin-Madison

[7] Chance Reschke, Thomas Sterling, Daniel Ridge, *A Design Study of Alternative Network Topologies for the Beowulf Parallel Workstation*, NASA Goddard Space Flight Center

[8] Frederick Douglis, *Transparent Process Migration in the Sprite Operating System*, University of California, Berkeley, 1990

[9] Daniel P. Bovet, Marco Cesati, *Understanding the Linux Kernel*, O'Reilly, 2001

[10] Alessandro Rubini, *Linux Device Drivers*, O'Reilly, 1998

[11] Andrew S. Tannenbaum, *Modern Operating Systems*, Prentice Hall, 2001

[12] Barak, Guday, Wheeler, *The MOSIX Distributed Operating System*, Springer Verlag, 1993

[13] Jim Pruyne, Miron Livny, *Managing Checkpoints for Parallel Programs*, University of Wisconsin-Madison

[14] Compaq, *Single System Image Clusters for Linux*, http://sourceforge.net/projects/ssic-linux

[15] David Gibson, *esky - A slightly portable user-space checkpointing system*, 2001

[16] William R. Dieter, James E. Lumpp, *User-level Checkpointing for LinuxThreads Programs*, University of Kentucky, 2001

[17] IEEE CS Task Force on Cluster Computing, *Single System Image* http://www.clustercomp.org

[18] Compaq, *Cluster Infrastructure for Linux* http://sourceforge.net/projects/ci-linux

[19] IEEE Std. 1003.1b-1993, *Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API)*, Institute of Electrical and Electronics Engineers, Inc., 1994

[20] Wensong Zhang, *Linux Virtual Server for Scalable Network Services*, National Laboratory for Parallel & Distributed Processing, China, http://www.LinuxVirtualServer.org

[21] IBM, *ESA/390 Principles of Operation*, http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/DZ9AR005/CCONTENTS?SHELF=EZ2HW119, 1998

[22] Michael Holzheu, *Linux VIPA*, IBM Research Report

[23] http://linux-ha.org, *Setting up Redundant Networking*,

[24] OpenGFS, http://www.opengfs.org

[25] IBM, *Linux for S/390: Device Drivers and Installation Commands*, 2001

[26] IBM, *Linux for S/390: ELF Application Binary Interface Supplement*, 2001

[27] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, J. Walpole, *MPVM: A Migration transparent version of PVM*, Department of Computer Science and Engineering Oregon Graduate Institute of Science & Technology, 1995

[28] S. Zhou, J. Wang, X. Zheng, P. Delisle, *UTOPIA: A load sharing facility for large, heterogeneous distributed Computer Systems*, Technical report, 1992

[29] http://www.sistina.com/gfs_howtos/, *GFS Howto*, 2001

[30] Georg Stellner, *Consistent Checkpoints of PVM Applications*, Institut für Informatik der Technischen Universität München,

[31] James S. Plank, Micah Beck, Gerry Kingsley, Kai Li, *Libckpt: Transparent Checkpointing under UNIX*, Usenix Winter 1995 Technical Conference, 1995

[32] http://oss.sgi.com/projects/failsafe/, *Linux FailSafe*

[33] A. L. Beguelin, J. J. Dongarra, A. Geist, R. J. Manchek, S. W. Otto, J. Walpole, *PVM: Experiences, current status and future direction*, Supercomputing ’93 Proceedings, 1993

[34] Shu Zhang, Mujtaba Khambatti, Partha Dasgupta, *Process Migration through Virtualization in a Computing Community*, Arizona State University