Unabhängiges Checkpointing in einer heterogenen Grid-Umgebung

Masterarbeit

von

Eugen Feller

aus

Düsseldorf

vorgelegt an der

Abteilung Betriebssysteme
Prof. Dr. Michael Schöttner
Heinrich-Heine-Universität Düsseldorf

19. November 2009

Gutachter:
Prof. Dr. Michael Schöttner
Dr. Christine Morin

Danksagung

An dieser Stelle möchte ich mich bei meinen Gutachtern Prof. Dr. Michael Schöttner und Dr. Christine Morin bedanken. Prof. Dr. Michael Schöttner danke ich für das Thema selbst und die hervorragende Unterstützung während der gesamten Zeit. Dr. Christine Morin danke ich ebenfalls für ihre hervorragende Unterstützung und die prompte Bereitschaft die Arbeit von Frankreich aus zu betreuen.

Ein ganz besonderer Dank gilt meinem Betreuer Dipl.-Inf. John Mehnert-Spahn für seine zahlreichen Tipps, Ideen und Anregungen zur inhaltlichen und strukturellen Optimierung dieser Arbeit.

Außerdem bedanke ich mich bei Thomas Ropars, M.Sc. der mir während meines Auslandssemesters in Frankreich bei INRIA stets mit gutem Rat und Tat zur Seite stand.

Meiner Familie danke ich für die Unterstützung während meines gesamten Studiums. Ohne sie wäre diese Arbeit nicht möglich gewesen. Ein besonderer Dank gilt meinem Bruder Alexander Feller, der mir dabei half einige sprachliche und logische Unstimmigkeiten aufzudecken.

Ein abschließender Dank gilt allen, die Interesse für diese Arbeit zeigten und mich mit ihren Vorschlägen unterstützten.

Abstract

The topic of this master thesis is related to the EU funded XtreemOS project. The aim of this project is to implement a Linux-based open source grid operating system. Grids in general allow the execution of many different kinds of applications e.g. client-server applications, scientific applications and many others. Hence, they are vulnerable to different kinds of failures either from the outside caused by node failures or inside by programming errors. Checkpoint-based fault tolerance saves the state of these applications to stable storage periodically under the so-called "checkpoint". In the event of failure the application state is rolled back according to the last taken checkpoint. Thus, upon failure the computation can be resumed from the last consistent checkpoint without the loss of the complete computations.

In order to achieve fault tolerance for distributed applications on the grid-level XtreemOS deploys a heterogeneous grid checkpointing service called XtreemGCP. This service is in charge of coordinating the checkpoint by addressing the underlying grid-node checkpointers (e.g. BLCR, LinuxSSI, OpenVZ, DMTCP and many others) in a transparent manner through a so called "Common Kernel Checkpointer API".

The aim of this master thesis is to develop and integrate the uncoordinated checkpointing strategy into the XtreemGCP service in order to allow the applications to take checkpoints independently. Thus, the processes of the application are able to decide on their own when it is the most suitable time to take a checkpoint. Due to the heterogeneous grid environment consisting of grid-nodes with different underlying checkpointers the focus of this master thesis relies on developing a solution which is able to operate independently of the underlying checkpointers.

As as starting point various theoretical backgrounds of XtreemOS and checkpointing in general are studied in the second chapter of this thesis. The following chapters introduce the design and implementation of the independent checkpointing strategy in the context of XtreemOS. At first the assumptions which were made during this thesis are introduced. Furthermore, it is necessary for the processes to record the dependencies among their checkpoints during failure-free execution in order to generate a global consistent state, called recovery-line during the recovery. Therefore a mechanism is introduced which is in charge of detecting and storing this dependency information in form of determinants on a persistent storage (e.g. XtreemFS). In order to calculate the recovery-line a rollback propagation algorithm has been described and integrated into the XtreemGCP service. This algorithm makes use of the dependency information to generate a global consistent state which is later used to restart application to the appropriate version.

Because every process of an application is allowed to take checkpoints independently a dynamically loaded library has been developed in order to provide a checkpointing interface to the application

developer and send the checkpoint request to the XtreemGCP.
During the actual recovery the processes of a distributed application need to be restored according to the information from the calculated recovery-line. Hence, some processes may act either as client or server or both. Especially the case in which both act as client and server is the most common one since a distributed calculation usually implies information exchange in both directions. In order to restart such an application two approaches have been developed during this master thesis. The first approach restricts the processes to act either as servers or as clients and restarts the servers before the clients. Thus, it is impossible for a client to connect to a non existing server during the recovery under normal circumstances. The second approach does not make any restrictions on the role of the processes and thus is able to restart processes which act either as client or server or both.
The recovery itself can be triggered either by the user or by the XtreemGCP-Service itself and is currently able to deal with two scenarios. The first scenarios assumes that all processes of a distributed application have exit either because of internal errors or grid-node failures before initiating the recovery. In fact such an assumption is very uncommon in a large grid-environment. For this reason a second scenario has been defined and integrated into the XtreemGCP. This scenario detects the failed processes based on the information from a dedicated failure monitor and restarts these processes only. Consider that during the recovery, already running processes might need to do a rollback to some earlier version according to the calculated recovery line. The design and implementation is able to deal with such a case and would terminate the running processes before doing the actual rollback.
The work done during this master thesis has been fully integrated into the XtreemGCP service. Furthermore, the implementation has been performance evaluated within a heterogeneous grid environment consisting of six nodes (BLCR and LinuxSSI) with a custom client-/server application. The results show that independent checkpointing with BLCR is slower in 4 orders of magnitude compared to LinuxSSI because of internal BLCR reasons. The overhead for the dependency tracking is under one microsecond. Moreover the recovery duration of the test application grows linearly with the number of processes.
The performance evaluation has shown the capability of the implementation to independently checkpoint a distributed application within a heterogeneous grid environment. This proof of concept can be further improved and extended with other functionality in the future (e.g. automated recovery, message-logging, garbage collection and more).

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

AEM	Application Execution Manager
API	Application Programming Interface
ExecMng	Execution Manager
GLIBC	GNU C Library
HHU	Heinrich-Heine-Universität
IPC	Interprozesskommunikation
JobDir	Job Directory
JobMng	Job Manager
JSDL	Job Submission Description Language
MJC	MultiJobControl
NFS	Network File System
PWD	Piecewise Deterministic
PXE	Preboot eXecution Environment
RecMng	Recovery Manager
SSI	Single System Image
VO	Virtuelle Organisation
XtreemGCP	XtreemOS Grid-Checkpointer
XtreemOS-F	XtreemOS Foundation
XtreemOS-G	XtreemOS Grid

Kapitel 1

Einleitung

1.1 Motivation

Aufgrund des ständig wachsenden Bedarfs an gemeinsam nutzbarer Rechenleistung, wurden Grid-Systeme in den letzten Jahren immer mehr zu einem wichtigen Forschungsthema [FK99]. Sie verwenden offene Schnittstellen und zeichnen sich durch die lose Kopplung, Heterogenität und die Möglichkeit einer geografischen Zerstreuung der Computer aus. Mittels virtueller Organisationen (VOs) geben sie außerdem Unternehmen die Möglichkeit sich zu vereinen, um an gemeinsamen Projekten zu arbeiten. Dem Benutzer gegenüber erscheinen sie dabei als ein einzelnes, großes, virtuelles System.
Es existieren verschiedene Ansätze Grid-Technologien zu entwickeln. Bei dem am meisten erforschten Middleware-Ansatz übernimmt eine zusätzliche Schicht die Verwaltung der Ressourcen des nativen Betriebssystems. Ein alternativer Ansatz ist die Verlagerung der wichtigen Gridfunktionalitäten in das Betriebssystem.
Das von der EU geförderte XtreemOS-Projekt [CFJ+08] setzt sich genau das zum Ziel. XtreemOS ist ein Linux-basiertes Grid-Betriebssystem welches um VO-Verwaltung und Schnittstellen für andere Grid-Dienste erweitert wurde. Im Gegensatz zu einem Middleware Ansatz verhält sich XtreemOS dem Benutzer gegenüber ähnlich wie ein traditionelles Betriebssystem. Dabei bleibt die gesamte Komplexität zur Verwaltung der Ressourcen vor dem Benutzer verborgen. Die für die Ausführung einer Anwendung notwendigen Ressourcen werden automatisch mittels der POSIX und SAGA-Schnittstellen im Grid transparent angefordert und verwaltet.
Aufgrund der hohen Anzahl der vorhanden Knoten, sind Grids dafür prädestiniert, verteilte

Anwendungen auszuführen, die über einen langen Zeitraum hinweg komplexe Berechnungen durchführen. Da die Menge der Knoten jedoch groß sein kann, sind Ausfälle so gut wie unvermeidbar. Zusätzlich können Anwendungen auch aufgrund von internen, unvorhersehbaren Ereignissen ausfallen. Demzufolge bedarf es an Mechanismen, die sich um die Wiederherstellung der ausgefallenen Anwendungen kümmern. Dadurch können Ausfälle toleriert und der fehlerfreie Betrieb gewährleistet werden.
Rollback-Recovery [EAWJ99] bietet in diesem Zusammenhang die Möglichkeit fehlerfreie Ausführung der langläufigen Anwendungen sicher zu stellen. Hierbei wird der Anwendungszustand unter dem sogenannten „Checkpoint“ in periodischen Abschnitten auf einem persistenten Speicher vorgehalten. Im Fehlerfall kann die Berechnung von dem letzten konsistenten „Checkpoint“ aus, ohne den kompletten Verlust der Daten, fortgesetzt werden.
Da sich die verteilten Anwendungen in einer Grid-Umgebung über mehrere unterschiedliche Grid-Knoten mit jeweils unterschiedlichen Checkpointern (z.B. BLCR, LinuxSSI, OpenVZ, DMTCP, Libckpt, Condor, etc.) erstrecken können, bedarf es eines verteilten Dienstes auf der Grid-Ebene, der all diese Checkpointer verwaltet. Im Rahmen von XtreemOS übernimmt diese Aufgabe ein Grid-Checkpointer (XtreemGCP), der dafür verantwortlich ist, Fehlertoleranz auf der Grid-Ebene zu gewährleisten.

1.2 Ziel der Arbeit

In dieser Arbeit soll eine Lösung für unabhängiges Checkpointing ohne Nachrichtenaufzeichnung entworfen und in die XtreemGCP Komponente integriert werden. Da es sich bei dem XtreemGCP um einen verteilten Dienst handelt, der mehrere unterschiedliche Grid-Knoten (PC, Cluster, etc.) anspricht, wird der Schwerpunkt dieser Arbeit auf Heterogenität gelegt. Das Ziel dieser Arbeit ist die Entwicklung einer Lösung zur unabhängigen Sicherung und Wiederherstellung der auf den unterschiedlichen Grid-Knoten laufenden, verteilten Anwendungen.
Dabei sollen primär zwei Komponenten entwickelt werden, die eine konsistente Sicherung und Wiederherstellung solcher Anwendungen erlauben. Da im Rahmen dieser Arbeit ausschließlich verteilte Anwendungen betrachtet werden, die mittels Nachrichtenaustausches kommunizieren, besteht die erste Komponente aus einer gemeinsam genutzten Bibliothek, die für die Aufzeichnung der Abhängigkeiten innerhalb der Anwendung zuständig ist.

Darüber hinaus stellt sie dem Programmierer eine Schnittstelle zur Verfügung, um einen Checkpoint aus der Anwendung heraus auszulösen. Die zweite Komponente befindet sich auf der Grid-Ebene und ist dafür verantwortlich, Ausfälle innerhalb der Anwendung zu erkennen, einen konsistenten Zustand zu berechnen und die ausgefallenen Teile zu einem geeigneten Zeitpunkt wiederherzustellen.
Abschließend soll die Implementierung auf ihre Performance hin untersucht werden.

1.3 Struktur der Arbeit

Diese Arbeit ist in fünf Kapitel „Grundlagen, Entwurf und Implementierung des unabhängigen Checkpointings, Leistungsbewertung und Zusammenfassung“ unterteilt.

Kapitel 2 befasst sich mit den, für das weitere Verständnis der Arbeit notwendigen theoretischen Grundlagen. Zunächst wird ein Einblick in die Architektur des XtreemOS-Projekts gegeben. Es folgt eine kurze Einführung in die Grundlagen der Checkpointing-Thematik. Abschließend wird die XtreemGCP Komponente des XtreemOS-Projekts ausführlich erläutert.

Kapitel 3 widmet sich dem Entwurf eines Konzepts zum unabhängigen Sichern einer verteilten Anwendung. Dabei wird zunächst die Aufzeichnung der Abhängigkeiten und die anschließende Berechnung eines für die Wiederherstellung notwendigen, konsistenten Zustandes beschrieben. Im weiteren Verlauf des Kapitels wird ein Konzept zum unabhängigen Sichern einer Anwendung vorgestellt. Dies beinhaltet sowohl die Erstellung von unabhängigen Sicherungen, als auch die Wiederherstellung von ausgefallenen Komponenten.

Kapitel 4 beschäftigt sich mit der Implementierung des in Kapitel 3 vorgestellten Konzepts. Dabei werden die entwickelten Komponenten und deren Integration in die bestehende XtreemGCP Architektur ausführlich erläutert. Das nachfolgende Kapitel 5 widmet sich der Performance-Evaluierung.

Zuletzt werden in Kapitel 6 die Ergebnisse dieser Arbeit zusammengefasst und ein kurzer Ausblick in weiterführende Entwicklungsmöglichkeiten gegeben.

Kapitel 2

Grundlagen

Dieses Kapitel behandelt die für das weitere Verständnis der Arbeit wichtigen Grundlagen aus dem Themengebiet XtreemOS und Checkpointing. Es beginnt mit einer kurzen Einführung in das XtreemOS Projekt gefolgt von einer Einführung in die Grundlagen der Checkpointing-Thematik. Beim letzteren werden die grundlegenden Checkpoint-Protokolle inklusive deren Vor- und Nachteile vorgestellt. Abschließend wird die Architektur der XtreemGCP Komponente ausführlich erläutert.

2.1 XtreemOS: Grid-Betriebssystem

In Kapitel 1 wurde bereits eine kleine Einführung in das XtreemOS Projekt gegeben. In diesem Abschnitt geht es primär um die Architektur von XtreemOS.
XtreemOS ist ein auf Linux basierendes Grid-Betriebssystem, welches aus einer Vielzahl von Kernel- und Userlevel-Komponenten zusammengesetzt ist. All diese Komponenten werden in der Abbildung 2.1 dargestellt. Es wird grundsätzlich zwischen zwei Schichten unterschieden.
Auf der obersten Schicht XtreemOS Grid (XtreemOS-G) [CFJ+08] befinden sich die High-level Grid-Dienste. Sie sind unter anderem für die Ausführung von Anwendungen, Auffindung und Verwaltung von Ressourcen, VOs und Benutzern verantwortlich. All diese Dienste können mittels einer POSIX und SAGA-Schnittstelle angesprochen werden.
Auf der untersten Schicht XtreemOS Foundation (XtreemOS-F) befinden sich die unterschiedlichen XtreemOS Ausführungen mit den notwendigen Linux-Erweiterungen

(Kernel Module, Userspace Bibliotheken, etc.). Dabei gibt es derzeit eine Ausführung für gewöhnliche PCs, Cluster von Linux-Rechnern und eine für mobile Endgeräte.
Die PC Version heißt Linux-XOS und bringt eine vollwertige VO-Unterstützung mit sich. Die SSI-Clusterausführung von XtreemeOS wird als LinuxSSI bezeichnet. Es handelt sich dabei um eine Erweiterung des Linux-basierten Kerrighed Betriebssystems für die VO und Grid-Komponenten Unterstützung. Kerrighed ist ein Single System Image (SSI) Betriebssystem, welches den Cluster als einen logischen, leistungsfähigen Grid-Knoten darstellt. Die mobile Ausführung genannt XtreemOS-MD beinhaltet ebenfalls eine vollwertige VO-Unterstützung und eine Reihe von Diensten zur Ausführung der Anwendungen, zum Datenzugang und zur Verwaltung von Benutzerkonten. Im Gegensatz zu allen anderen XtreemOS Ausführungen stellen mobile Endgeräte aus Mangel an Rechenleistung, deren Ressourcen nicht dem Grid zur Verfügung und agieren lediglich als Clients.

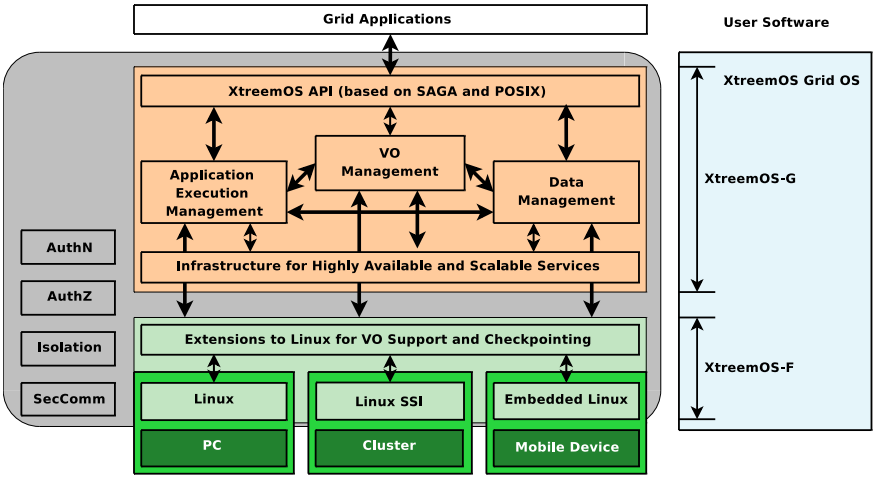

Abbildung 2.1: XtreemOS Architektur [CFJ+08]

2.2 Checkpointing Grundlagen

2.2.1 Der Domino-Effekt

Beim Checkpointing wird der Zustand einer verteilten Anwendung in periodischen Abschnitten auf einem persistenten Speicher in Form eines Checkpoints festgehalten. Dadurch kann die Anwendung im Fehlerfall in einen konsistenten Zustand wiederhergestellt werden.

Es existieren mehrere Möglichkeiten einen global konsistenten Zustand einer verteilten Anwendung zu erzeugen. Während beim koordinierten Checkpointing ein Initiator-Prozess dafür eingesetzt wird, den Checkpoint-Vorgang der Prozesse zu koordinieren, ist beim unabhängigen Checkpointing jeder Prozess in der Lage selbstständig Sicherungen zu einem beliebigen Zeitpunkt vorzunehmen. Bei der letzteren Methode wird der global konsistente Zustand, genannt Recovery-Line, erst bei der Wiederherstellung der Anwendung anhand der zuvor aufgezeichneten Abhängigkeiten ermittelt. Dadurch kann bei dieser Methode der so genannte Domino-Effekt [Abb88] auftreten. Dies ist immer dann der Fall, wenn keine Recovery-Line auf Basis der aufgezeichneten Determinanten berechnet werden konnte und damit ein Rückfall in den initialen Zustand unvermeidbar ist. Die Folge ist ein Datenverlust, der im schlimmsten Fall so weit gehen kann, dass die Anwendung eine komplette Berechnung erneut durchführen muss.
Ein solcher Fall wird in der Abbildung 2.2 dargestellt. Es handelt sich hierbei um eine verteilte Anwendung bestehend aus zwei Prozessen P1 und P2, die unabhängig voneinander Sicherungen vornehmen. Dabei werden die Nachrichten durch Pfeile, und die Sicherungen durch blaue Punkte dargestellt.

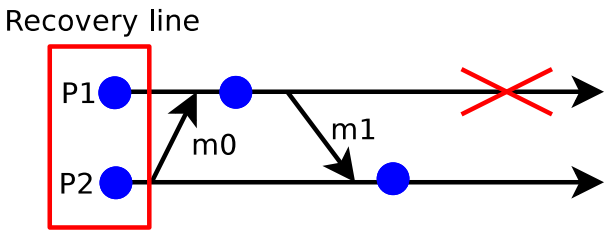

Abbildung 2.2: Domino-Effekt

Sowohl P1 als auch P2 starten deren Ausführung mit einem initialen Checkpoint. Nach dem zweiten Checkpoint sendet P1 eine Nachricht an P2. Die gesendete Nachricht wird von P2 empfangen und in einer Determinante aufgezeichnet. Anschließend erfolgt aufgrund des Ausfalls von P1 eine Berechnung der Recovery-Line. Da der zweite Checkpoint von P2 eine Nachricht enthält, die nicht im zweiten Checkpoint von P1 erhalten ist, muss P2 auf den ersten Checkpoint zurückrollen. Nach diesem Vorgang erhält wiederum der zweite Checkpoint von P1 eine Nachricht, die nicht im ersten Checkpoint von P2 enthalten ist. Dadurch muss P1 ebenfalls auf den ersten Checkpoint zurückrollen. Das Ergebnis der Berechnung ist Recovery-Line, die zwar konsistent ist, aber gleichzeitig den initialen Zustand der gesamten Anwendung widerspiegelt.

2.2.2 Log-basierte Rollback-Recovery

Die Log-basierte Rollback Recovery-Strategie umfasst die Erstellung der Checkpoints und die Aufzeichnung der eigentlichen Nachrichtenübertragung [BCH+03]. Dadurch ist es im Fehlerfall möglich, sowohl eine konsistente als auch eine deterministische Wiederherstellung der verteilten Anwendung durchzuführen. Während der Wiederherstellungsphase werden die Nachrichten in der selben Reihenfolge eingespielt in der sie empfangen wurden.
Aufgrund der Nachrichtenaufzeichnung und Wiedereinspielung ist die Log-basierte Rollback Recovery-Strategie resistent gegenüber dem, in Abschnitt 2.2.1 vorgestellten, Domino-Effekt und erlaubt es außerdem Checkpoints unabhängig erstellen zu können.
Log-basierte Rollback Recovery-Strategie verlässt sich auf das stückweise, deterministische Ausführungsmodell (PWD) [DG96]. Dieses Modell besagt im Wesentlichen, dass die Prozessausführung als eine Sequenz von Intervallen mit deterministischen Zuständen modelliert werden kann, wobei jedes Intervall mit einem nichtdeterministischen Ereignis beginnt. Unter einer solchen Annahme identifiziert die Log-basierte Rollback Recovery-Strategie alle nichtdeterministischen Ereignisse und hält diese in Form von einer Determinanten auf einem persistenten Speicher fest. Diese Information kann zur Wiedereinspielung aller Ereignisse während des Wiederherstellungsvorgangs verwendet werden. Dadurch ist die Log-basierte Rollback Recovery-Strategie in der Lage eine Anwendung im Fehlerfall unter der Bewahrung der Ausführungsreihenfolge wiederherzustellen.

2.2.3 Checkpoint-basierte Rollback-Recovery

2.2.3.1 Übersicht

Beim der Checkpoint-basierten Rollback Recovery-Strategie wird die Anwendung wiederhergestellt, indem sie in den letzten konsistenten Zustand versetzt wird. Im Gegensatz zu der Log-basierten Rollback Recovery-Strategie verlässt sie sich nicht auf das PWD-Modell und muss somit keine nichtdeterministischen Ereignisse erkennen, aufzeichnen und während der Wiederherstellung erneut einspielen. Dadurch garantiert sie zwar nach einem Ausfall einen konsistenten Zustand, jedoch keine deterministische Ausführung der Anwendung.

Es wird grundsätzlich zwischen zwei Checkpoint-basierten Strategien unterschieden: Unabhängiges und koordiniertes Checkpointing. Eine dritte Strategie ist das kommunikationsinduzierte Checkpointing. Sie ist aufgrund ihrer ungenügenden Skalierung bei einem hohen Nachrichtenaufkommen für diese Arbeit nicht relevant und wird nicht näher betrachtet.

2.2.3.2 Unabhängiges Checkpointing

Bei unabhängigem Checkpointing [EAWJ99] wird den Prozessen die Möglichkeit gegeben, selbst zu entscheiden, zu welchem Zeitpunkt eine Sicherung erstellt werden soll. Dafür müssen während der gesamten Ausführungszeit der Anwendung alle nichtdeterministischen Ereignisse (z.B. Empfang von Nachrichten) auf einem persistenten Speicher festgehalten werden. Diese Informationen werden im Fehlerfall dazu verwendet, einen global konsistenten Zustand der Anwendung zu berechnen.
Der wesentliche Vorteil dieser Methode liegt an der Flexibilität, die Checkpoints zu einem am besten geeigneten Zeitpunkt vornehmen zu können. Dadurch hat ein Prozess die Möglichkeit, den Checkpoint-Overhead zu vermindern, indem die Sicherungen zu den Zeitpunkten vorgenommen werden, an denen die zu sichernde Datenmenge gering ist [WCLF95].
Dennoch hat dieses Verfahren einige Nachteile. Ohne die Kombination mit einem Log-basierten Protokoll ist es anfällig gegenüber dem in Abschnitt 2.2.1 erläuterten Domino-Effekt. Zusätzlich, zur Problematik des Domino-Effekts, können von einem Prozess Sicherungen vorgenommen werden, die niemals ein Teil des globalen Zustandes werden. Sie sind somit nicht in die Berechnung der Recovery-Line involviert und belegen lediglich Festplattenspeicher. Aufgrund der vielen Checkpoints, die von einem einzelnen Prozess erstellt werden können, bedarf unabhängiges Checkpointing schließlich der Ausführung eines Garbage-Collection Algorithmus, um die nicht benötigten Sicherungen zu entfernen.

2.2.3.3 Koordiniertes Checkpointing

Im Gegensatz zum unabhängigen Checkpointing übernimmt beim koordinierten Checkpointing ein Initiator-Prozess die Koordinierung der Prozesse, um eine global konsistente

Abbildung der Anwendung zu erstellen. Dies hat zur Folge, dass immer global konsistente Checkpoints erzeugt werden. Außerdem ist koordiniertes Checkpointing resistent gegenüber dem Domino-Effekt. Im Falle eines Absturzes starten alle Prozesse vom jeweils letzten Checkpoint wieder neu. Zusätzlich entfällt die Berechnung der Recovery-Line so wie die Notwendigkeit einer Garbage-Collection.
Es existieren zwei unterschiedliche Vorgehensweisen, um einen global konsistenten Checkpoint vorzunehmen:

1. Bei einem blockierenden Checkpoint-Protokoll werden zunächst alle Prozesse während der Koordinierungsphase angehalten und die Nachrichtenkanäle gespült. Anschließend wird jeder Prozesse aufgefordert einen Checkpoint vorzunehmen. Nach einer erfolgreichen Sicherung setzten die Prozesse ihre Ausführung fort. Ein solches Vorgehen hat einen großen zeitlichen Overhead und ist nicht zu empfehlen.
2. Bei einen nicht blockierenden Checkpoint-Protokoll erstellt der Initiator-Prozess einen Checkpoint und versendet einen Marker an alle beteiligen Prozesse. Sobald ein Prozess diesen Marker empfängt, löst er seinerseits einen Checkpoint aus und leitet diesen Marker an alle Prozesse weiter die ihrerseits das gleiche machen. Dabei ist es wichtig zu beachten, dass der Marker die höchste Priorität hat und damit vor allen anderen Anwendungsnachrichten [EAWJ99] versendet wird.

An dieser Stelle ist es wichtig zu erwähnen, dass bei einem nicht-blockierenden Checkpoint Protokoll diverse Interprozesskommunikation (IPC) - Ressourcen (Semaphoren, Message Queues und Shared Memory) so wie Dateien, während der Sicherung verändert werden können. Dadurch können inkonsistenten Auftreten, die unter der Verwendung eines blockierenden Checkpoint-Protokolls nicht aufgetreten wären. Dies hängt damit zusammen, dass dieses Protokoll nur auf die Nachrichtenebene ausgerichtet ist.

2.3 XtreemGCP Architektur

Dieser Abschnitt behandelt die Checkpointing-Komponente (XtreemGCP) von XtreemOS. Sie ist ein wichtiger Bestandteil des Projekts und wird, wie bereits in Kapitel 2 erläutert, dafür eingesetzt um Migration und Fehlertoleranz für Jobs auf der Grid-Ebene zu erreichen. Dafür verwaltet sie mehrere unterschiedliche Checkpointer (z.B. BLCR [DHR03],

LinuxSSI [Xtr07], OpenVZ, etc.) und implementiert die in Abschnitt 2.2.3 vorgestellten Checkpoint-Strategien, wobei in der aktuellsten Version ausschließlich koordiniertes Checkpointing unterstützt wird. Die Erweiterung für unabhängiges Checkpointing ist das Ziel dieser Arbeit und wird in Kapitel 3 ausführlich vorgestellt.

2.3.1 Terminologie

An dieser Stelle werden wichtige Begriffe erläutert, die im Laufe dieser Arbeit verwendet werden. Jeder Job wird im Grid durch eine eindeutige JobID identifiziert. Im Falle einer verteilten Anwendungen besteht jeder Job aus mehreren, sogenannten Job-Units. Jede Job-Unit enthält eine Menge von Prozessen, die durch eine Prozessgruppen-ID [SRSM08] gekennzeichnet sind.
Um die Ausführung und Verwaltung der Jobs kümmern sich der sogenannte Application Execution Manager (AEM). Diese verteilte Komponente besteht aus mehreren Diensten, die jeweils unterschiedliche Aufgaben übernehmen. Solche Aufgaben reichen in der Regel von der Ressourcen bis hin zur VO-Verwaltung.
Für die Job-Verwaltung ist der Job-Manager (JobMng) Dienst zuständig. Dieser verteilte Dienst beantwortet Anfragen auf der Grid-Ebene, die sich an die Jobs richten. Solche Anfragen können z.B. Job-Informationen, Scheduling-Anweisungen, Koordinierung des Checkpoints [SRSM08] oder auch Migrationsentscheidungen sein.
Jeder Job wird von genau einem Job-Manager verwaltet. Um das Auffinden der Jobs zu ermöglichen, verwaltet ein weiterer verteilter Dienst, genannt Job-Directory eine jeweils aktuelle Liste der laufenden Jobs und deren Ausführungsorte.
Für die lokale Ausführung und Verwaltung der Job-Units auf dem jeweiligen Grid-Knoten ist der Execution-Manager (ExecMng) zuständig. Seine Aufgabe besteht im Empfang und in der Ausführung der Anweisungen der jeweiligen Job-Manager. Solche Anweisungen können zum Beispiel Signale, oder auch die Ausführung von Checkpoint/Migrations [SRSM08] Anforderungen sein, die an eine Job-Unit weitergeleitet werden.
Für die persistente Haltung der Checkpoint-Daten und Job-Metadaten wird das im Rahmen des XtreemOS-Projekts einwickelte, verteilte Dateisystem XtreemFS verwendet.

2.3.2 Grid-Checkpointer

Wie bereits in Abschnitt 2.3 erläutert, ist auf der Grid-Ebene der XtreemGCP-Dienst für die Überwachung des Sicherungs- und Wiederherstellungsverhaltens verantwortlich. Dieser Dienst ist in mehrere Schichten unterteilt, wobei primär zwischen der Grid- und der Node-Ebene unterschieden wird (siehe Abbildung 2.3).

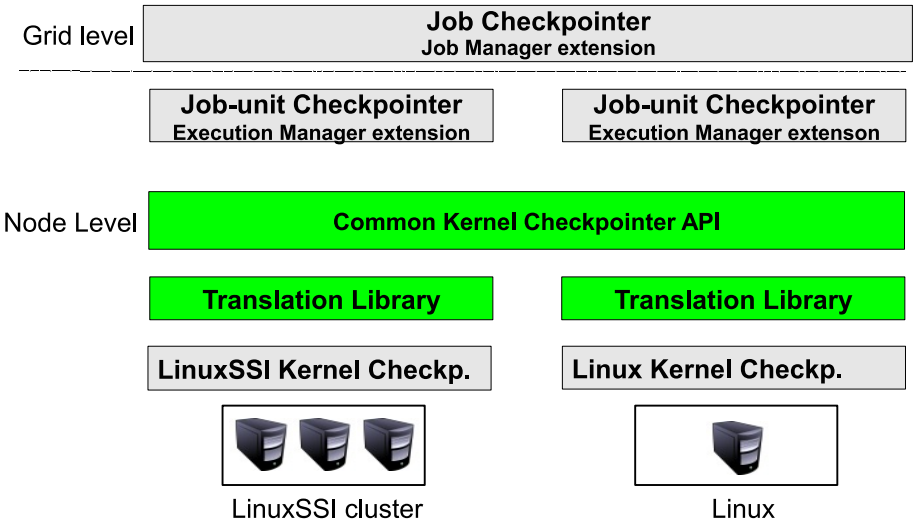

Abbildung 2.3: XtreemOS Grid-Checkpointer Architektur [SRSM08]

Auf der Grid-Ebene übernimmt der Job-Checkpointer die Sicherung und Wiederherstellung der Job-Units. Dafür spricht er die auf der Node-Ebene liegenden Job-Unit Checkpointer an, um einen Checkpoint/Restart der jeweiligen Job-Units vorzunehmen. Der Job-Unit Checkpointer adressiert auf einer transparenten Art und Weise die zugrunde liegenden Checkpointer.
Für die Ansteuerung der unterschiedlichen Checkpointer existiert ein Application Programming Interface (API) genannt „Common Kernel Checkpointer API“ [SRSM08]. Diese Schnittstelle wird vom Job-Unit Checkpointer dafür benutzt, um potentiell mehrere Checkpointer, auf einer vereinheitlichten Weise mittels einer geladenen Übersetzungsbibliothek (Translation-Library) anzusteuern. Die Übersetzungsbibliothek implementiert dieses Interface für den jeweiligen Checkpointer. Sie ist ebenfalls für die Übersetzung der Grid-Semantiken in die für einen Checkpointer verständlichen Semantiken verantwortlich.

2.3.2.1 Job Checkpointer

Der Job-Checkpointer ist eine der Kernkomponenten des XtreemGCP-Dienstes. Er befindet sich immer auf dem selben Knoten auf dem sich die Job-Manager-Instanzen eines Jobs betreffend befindet. Seine Aufgabe besteht darin, die Checkpoint-Strategien pro Job zu ermitteln und zu initialisieren, Meta-Daten anzulegen, Ressourcen anzufordern und einiges mehr.
Durch seine globale Sicht auf den Job kennt er stets alle Job-Units und deren Ausführungsorte. Damit ist er in der Lage die darunter liegenden Komponenten (Job-Unit Checkpointer) zu koordinieren, um unterschiedliche Checkpoint-Strategien umsetzen zu können. Im Falle koordinierten Checkpointings übernimmt er die Rolle des Koordinators. In diesem Kontext ist er dafür verantwortlich die Job-Unit Checkpointer zu synchronisieren, um einen global konsistenten Checkpoint eines Jobs erzeugen zu können. Während des Restarts übernimmt er ebenfalls die Koordinierung und sorgt dafür, dass der Job mit seinen Job-Units ordnungsgemäß wiederhergestellt wird.
Für die Erkennung der Job-Units und erneute Anforderung der verwendeten Ressourcen während der Wiederherstellung bedarf es dem Anlegen einiger Meta-Informationen während des Checkpoints. Das Ablegen dieser Informationen erfolgt ebenfalls vom Job-Checkpointer auf dem XtreemFS-Dateisystem [SRSM08].

2.3.2.2 Job-Unit Checkpointer

Jeder Grid-Knoten besitzt einen Job-Unit Checkpointer, der für die die Sicherung der lokalen Job-Units verantwortlich ist. Er wird während der Koordinierungsphase einer Sicherung vom Job-Checkpointer aufgerufen und spricht mit Hilfe der „Common Kernel Checkpointer API“ den ausgewählten Checkpointer an. Im Rahmen dieser Arbeit wurde seine Funktionalität erweitert. Somit ist es möglich mittels der Job-Units unabhängige Sicherungen anzulegen (mehr dazu in Kapitel 3).

2.3.2.3 Common Kernel Checkpointer API

Aufgrund der heterogenen Situation innerhalb eines Grids, besitzt jeder Grid-Knoten einen oder mehrere Checkpointer. Diese Checkpointer haben unterschiedliche Eigenschaf-

ten und Aufrufsemantiken. Um all diese Checkpointer einheitlich von der Grid-Ebene aus ansprechen zu können, wurde die bereits in Abschnitt 2.3.2 erwähnte „Common Kernel Checkpointer API“ entworfen. Diese API stellt dem Job-Unit Checkpointer eine einheitliche Schnittstelle zur Verfügung, um die jeweiligen Checkpointer anzusprechen. Außerdem bildet sie die Grid-Semantiken auf die Checkpointer-Semantiken [SRSM08] ab.

2.3.2.4 Translation Library

Die Translation Library implementiert die „Common Kernel Checkpointer API“ für den jeweiligen Checkpointer und wird dynamisch bei Bedarf (Checkpoint/Restart) vom zugehören Job-Unit Checkpointer geladen. Sie implementiert die für das Ansprechen des jeweiligen Checkpointers spezifische Routinen. Zusätzlich müssen für jeden Checkpointer bei Bedarf einige Vorbereitungen vor dem eigentlichen Checkpoint getroffen werden. Dies betrifft zum Beispiel das setzen von Capabilities oder Aufräumen von bereits bestehenden Checkpoint-Dateien. Die dafür notwendigen Routinen werden ebenfalls innerhalb der Translation Library zur Verfügung gestellt.
Derzeit bietet der XtreemGCP-Dienst Unterstützung für drei verschiedenen Checkpointer an, die mittels einer Translation Library eingebunden werden: BLCR, LinuxSSI und OpenVZ.

2.3.2.5 Callbacks

Die meisten derzeit verfügbaren Checkpointer unterstützen nur eine begrenzte Anzahl der für die Sicherung möglichen Ressourcen. Die restlichen Ressourcen (z.B. Sockets) können aufgrund von Checkpointer Einschränkungen nicht gesichert werden. Somit bedarf es zum einen einer Lösung um die nicht unterstützten Ressourcen zu sichern, zum anderen aber auch um bestimmte Ressourcen von der Sicherung auszuschließen. Dadurch lässt sich unter Umständen Zeit und Speicherplatz sparen.
Callbacks stellen dabei dem Programmierer ein Werkzeug zur Verfügung, um genau diesen Anforderungen gerecht zu werden. Sie werde entweder vom Anwendungsentwickler oder transparent mittels eines *fork*-Wrappers registriert und werden vor jeder Sicherung und/oder nach einer Wiederherstellung der Anwendung ausgeführt. Dadurch können

die vom Checkpointer nicht unterstützten Ressourcen gesichert werden. Mit Hilfe der Callbacks können zum Beispiel Dateien vor einem Checkpoint geschlossen werden. Außerdem ist es möglich die Socket-Verbindungen [Ste03] nach einer erfolgreichen Wiederherstellung der Anwendung wieder aufzubauen.

Kapitel 3

Konzeptueller Entwurf des unabhängigen Checkpointings

Wie bereits im Abschnitt 2.3 näher erläutert, unterstützt die derzeitige Implementierung des XtreemOS Grid-Checkpointers ausschließlich koordiniertes Checkpointing. Im Rahmen dieser Arbeit soll eine Version des unabhängiges Checkpointings ohne Nachrichtenaufzeichnung ausgearbeitet und in die bestehende XtreemGCP Komponente integriert werden. Da es sich bei dem XtreemOS Grid-Checkpointer um einen verteilten Dienst handelt, der mehrere unterschiedliche Checkpointer anspricht (z.B. BLCR, LinuxSSI, OpenVZ, etc.) liegt der Schwerpunkt dieser Arbeit auf der Heterogenität. Dadurch kann eine über mehrere Grid-Knoten laufende, verteilte Anwendung konsistent gesichert und wiederhergestellt werden (siehe Abschnitt 1.2).
Dieses Kapitel beschäftigt sich mit dem konzeptuellen Entwurf des unabhängigen Checkpointings im Kontext von XtreemOS. Zunächst werden in Abschnitt 3.1 die an das System gestellten Anforderungen eingeführt. Anschließend wird in Abschnitt 3.2 die Aufzeichnung der Abhängigkeiten innerhalb einer verteilten Anwendung näher erläutert. Schließlich wird in Abschnitt 3.3 ein Algorithmus zur Berechnung der Recovery-Line anhand der aufgezeichneten Abhängigkeitsinformationen vorgestellt.
Die nachfolgenden Abschnitte 3.4 und 3.5 beschreiben die Integration der Komponenten in die XtreemGCP Architektur. Zunächst wird in Abschnitt 3.4 ein Konzept zum unabhängigen Sichern einer verteilten Anwendung vorgestellt. Dieser Abschnitt beschäftigt sich mit der Aufzeichnung der Abhängigkeiten, Kommunikation mit dem Grid-Checkpointer und mit dem unabhängigen Erstellen der Sicherungen.
Anschließend wird im letzten Abschnitt 3.5 die Wiederherstellung einer verteilten An-

wendung beschrieben. In diesem Zusammenhang werden zunächst zwei Vorgehensweisen für die Wiederherstellung der Job-Units vorgestellt. Die erste Vorgehensweise geht davon aus, dass die Job-Units entweder als Client, oder als Server fungieren. Eine Kombination der Job-Units, die gleichzeitig als Client und Server agieren, wird bei diesem Ansatz nicht unterstützt. Die zweite Vorgehensweise hebt diese Einschränkung mit einem Koordinierungsprotokoll auf und ermöglicht dadurch die Wiederherstellung von Job-Units die gleichzeitig, sowohl als Client wie auch als Server operieren. Darüber hinaus werden in diesem Abschnitt zwei mögliche Wiederherstellungszenarien der verteilten Anwendung vorgestellt.
Im ersten Szenario findet die Wiederherstellung der verteilten Anwendung ausgehend von einem Totalausfall der Grid-Knoten statt. Das zweite Szenario geht von einem Teilausfall der Anwendung aus und stellt nur die ausgefallenen Komponenten wieder her. Für die Wiederherstellung der ausgefallenen Komponenten bedarf es eines Mechanismus zur Ausfallerkennung. Dieser Mechanismus wird ebenfalls im Rahmen dieses Abschnittes dargestellt.

3.1 Anforderungen

Im Rahmen dieser Arbeit werden ausschließlich verteilte Anwendungen betrachtet, die mittels Nachrichtenaustausches miteinander kommunizieren. Dabei besteht eine verteilte Anwendung aus *n*-Job-Units, die ihre Berechnungen untereinander mittels Eingabe-/Ausgabe-Nachrichten austauschen (siehe Abschnitt 2.3.1). Für die Nachrichtenübertragung wird ein zuverlässiges Übertragungsprotokoll (TCP) verwendet, welches die Nachrichten in einer First-In-First-Out (FIFO)-Reihenfolge ausliefert. Darüber hinaus können die Job-Units entweder als Client oder als Server fungieren. Eine Wiederherstellung von Job-Units, die gleichzeitig als Client und Server funktionieren, wurde konzeptuell ausgearbeitet und in Abschnitt 3.5.1 beschrieben.
Bei der ersten Ausführung macht jede Job-Unit zunächst einen initialen Checkpoint und setzt anschließend ihre Arbeit fort. Im Fehlerfall beenden sich die Job-Units nach dem Fail-Stop Modell [SS83]. Dabei werden die Ausfälle während des Checkpoint-Vorgangs nicht toleriert, da sie nichtdeterministisches Verhalten (z.B. Abstürze) im zuständigen Checkpointer auslösen können.
Darüber hinaus existiert ein verteiltes Dateisystem (z.B. XtreemFS), auf dem sowohl die

Abhängigkeitsinformationen, als auch die Sicherungen knotenübergreifend vorgehalten werden. Diese Informationen können im Fehlerfall zur Wiederherstellung des konsistenten Zustandes einer Anwendung benutzt werden.
Die Berechnung eines global konsistenten Zustandes kann entweder benutzerinitiiert oder automatisch, vom XtreemGCP-Dienst eingeleitet werden. Die Wiederherstellung wird ausgehend vom letzten Checkpoint aller Job-Units durchgeführt. Im Laufe dieses Vorgangs kommt der Rollback-Propagation Algorithmus [WF93] zum Einsatz. Abschnitt 3.3 befasst sich näher mit der Berechnung der Recovery-Line.

3.2 Aufzeichnen der Abhängigkeiten

Während der gesamten Ausführungszeit einer Anwendung müssen alle nicht deterministischen Ereignisse (z.B. Empfang von Nachrichten) auf einem persistenten Speicher für die spätere Berechnung der Recovery-Line festgehalten werden (siehe Kapitel 2). Dafür wird zunächst jeder beteiligten Job-Unit eine Bibliothek hinzugeladen. Diese Bibliothek ist schließlich für die Verwaltung und für die Aufzeichnung der Abhängigkeitsinformationen während der gesamten Laufzeit einer Job-Unit zuständig (siehe Abbildung 3.1). Mit Hilfe der aufgezeichneten Informationen, kann die Anwendung im Fehlerfall in einen konsistenten Zustand versetzt werden.

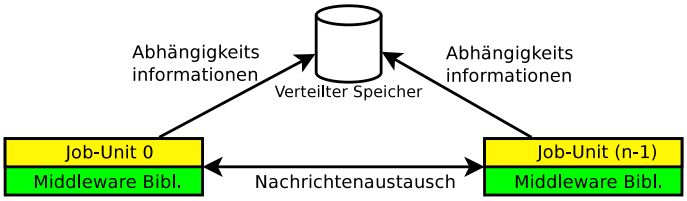

Abbildung 3.1: Aufzeichnen der Abhängigkeiten

Die Aufzeichnung der Abhängigkeiten erfolgt entsprechend der nachfolgenden Vorgehensweise. Sei $c_{i,x}$ der $x-te$ checkpoint $(x \geq 0)$ der Job-Unit JU_i $(0 \leq i \leq N-1)$. Dabei bezeichnen wir x als den Checkpoint-Index [WF93] und N als die Anzahl der vorhandenen Job-Units. Zwei Sicherungen $c_{i,x}$ und $c_{i,y}$ gelten als inkonsistent, wenn eine Nachricht nach $c_{i,x}$ gesendet und vor $c_{i,y}$ empfangen wurde. Darüber hinaus definieren wir $I_{i,x}$ als das Checkpoint-Intervall [WF93] zwischen den zwei Sicherungen $c_{i,x-1}$ und $c_{i,x}$. Sowohl der Checkpoint-Index als auch das Checkpoint-Intervall werden in der nachfolgenden Abbildung 3.2 näher veranschaulicht.

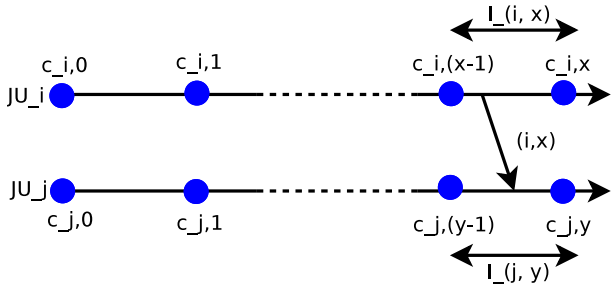

Abbildung 3.2: Checkpoint-Index und Checkpoint-Intervall

Dabei sendet die Job-Unit JU_i eine Nachricht an JU_j im Intervall $I_{i,x}$ und hängt der Nachricht das Paar (i,x) an. JU_j empfängt die Nachricht im Intervall $I_{j,y}$ und zeichnet eine Abhängigkeiten zwischen $c_{i,x-1}$ und $c_{i,y}$ auf.
Sollte die Job-Unit JU_j weitere Nachrichten im gleichen Intervall $I_{j,y}$ von derselben Job-Unit JU_i empfangen, so müssen keine weiteren Abhängigkeiten aufgezeichnet werden. Eine Nachricht ist also ausreichend, um die Abhängigkeit zu erkennen.

3.3 Berechnung der Recovery-Line

Für die erfolgreiche Berechnung eines global konsistenten Zustandes der verteilten Anwendung müssen die aufgezeichneten Abhängigkeitsinformationen aller Job-Units geladen werden, um die Recovery-Line zu berechnen. In diesem Abschnitt wird die Berechnung der Recovery-Line anhand der in der Abbildung 3.3 dargestellten Beispielanwendung erläutert. Es handelt sich hierbei um eine verteilte Anwendung, bestehend aus drei Job-Units, die mittels des Nachrichtenaustauschs miteinander kommunizieren. Nach dem Versenden jeder Nachricht wird eine Abhängigkeit mit dem in Abschnitt 3.2 vorgestellten Verfahren aufgezeichnet.

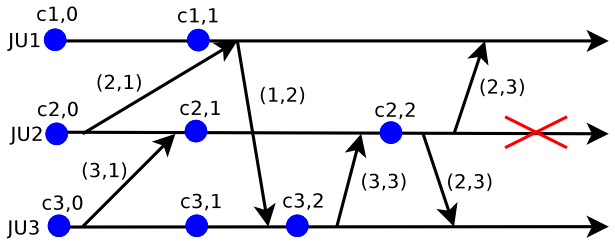

Abbildung 3.3: Verteilte Anwendung mit drei Job-Units

Im Falle eines Job-Unit Absturzes (zum Beispiel JU2) wird eine Recovery-Line berechnet. Für die Berechnung der Recovery-Line werden zunächst die aufgezeichneten Abhängigkeitsinformationen aller Job-Units ausgelesen. Mit Hilfe dieser Informationen erfolgt anschließend die Generierung eines Checkpoint-Graphen unter der Verwendung der nachfolgenden Regeln. Die Abbildung 3.4 zeigt den Checkpoint-Graph für die in Abbildung 3.3 dargestellte Anwendung. Die Recovery-Line ist mit Rot markiert
Eine gerichtete Kante wird von $c_{i,x-1}$ aus nach $c_{j,y}$ genau dann gezeichnet, wenn entweder $i \neq j$ ist und eine Nachricht von $I_{i,x}$ aus geschickt und in $I_{j,y}$ empfangen wurde, oder $i = j$ und $y = x + 1$ gilt.

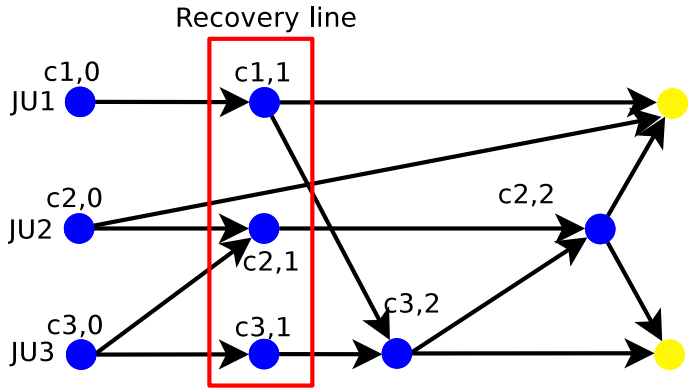

Abbildung 3.4: Checkpoint-Graph

Dabei werden zunächst die ausgefallenen Job-Unit Knoten aus dem Graphen entfernt und die restlichen noch laufenden Knoten Gelb markiert. Anschließend wird der Rollback-Propagation Algorithmus [WF93] auf den Checkpoint-Graph für die Berechnung der Recovery-Line angewandt (siehe Auflistung 3.1).

Listing 3.1: Rollback-Propagation Algorithmus

```
1 Füge den letzten Checkpoint von jeder ausgefallenen Job-Unit als Element in das
      RootSet hinzu.
2 Füge den aktuellsten Zustand von jeder noch laufenden Job-Unit als Element in das
      RootSet hinzu.
3 Markiere alle erreichbaren Checkpoints beim verfolgen von mindestens einer Kante
      ausgehenden von einem beliebigen Element im RootSet.
4 While (Mindestens ein Element im RootSet markiert)
5 Begin
6    Ersetze jedes markierte Element im RootSet mit dem letzten unmarkierten
         Checkpoint der selben Job-Unit.
7    Markiere alle erreichbaren Checkpoints beim verfolgen von mindestens einer
         Kante ausgehenden von einem beliebigen Knoten im RootSet.
8 End
```

Nach einer erfolgreichen Ausführung dieses Algorithmus beinhaltet der RootSet eine gültige Recovery-Line. Diese Recovery-Line wird im Fehlerfall dazu verwendet, die verteilte Anwendung in einen konsistenten Zustand zu versetzen.
Die in der Abbildung 3.4 in Rot dargestellte Recovery-Line veranschaulicht zwei wichtige Eigenschaften des unabhängigen Checkpointings ohne Nachrichtenaufzeichnung. Nach der Anwendung des Rollback-Propagation Algorithmus wurden alle Checkpoints ausgeschlossen, bei denen der Empfang einer noch nicht gesendeten Nachricht vorliegt. Solche Nachrichten werden auch als „Orphan-Messages“ bezeichnet. Dadurch wurde eine Menge von Checkpoints ermittelt, die einen konsistenten Zustand der Anwendung darstellt.
Darüber hinaus verdeutlicht die Abbildung 3.4 eine andere, wichtige Eigenschaft des unabhängigen Checkpointings ohne Nachrichtenaufzeichnung. Die Job-Unit *JU1* wird auf den Checkpoint *c1,1* zurückgerollt und erwartet den Empfang einer Nachricht, die der Sender *JU2* aufgrund der berechneten Recovery-Line *c2,1* nicht erneut verschicken wird. Eine solche Nachricht wird als „Lost-Message“ bezeichnet und kann im Kontext des unabhängigen Checkpointings ohne Nachrichtenaufzeichnung ignoriert werden, da hierbei keine deterministische Ausführung der Anwendung wie vor dem Fehler gewährleistet werden muss.

3.4 Durchführung einer unabhängigen Sicherung

Beim unabhängigen Checkpointing ist jede Job-Unit in der Lage Sicherungen eigenständig zu erzeugen. Dafür stellt die für die Aufzeichnung der Abhängigkeiten zuständige Middleware-Bibliothek dem Anwendungsentwickler eine spezielle Schnittstelle zur Verfügung. Mit Hilfe dieser Schnittstelle ist der Anwendungsentwickler in der Lage Sicherungen zu einem beliebigen Zeitpunkt vorzunehmen. Dabei besteht die Aufgabe dieser Schnittstelle in der Benachrichtigung des XtreemGCP. Dieser kümmert sich seinerseits um die Erstellung einer Sicherung und um die Aktualisierung der für eine Wiederherstellung der Anwendung notwendigen Job Meta-Informationen.
Nach dem Aussenden einer Middleware Checkpoint-Anfrage an den XtreemGCP wird diese von dem zuständigen Job-Unit Checkpointer verarbeitet. Der Job-Unit Checkpointer wählt den richtigen Checkpointer aus und erstellt daraufhin eine Sicherung der Anwendung. Im Falle einer erfolgreichen Sicherung aktualisiert er schließlich die sich auf einem verteilten Dateisystem (z.B. XtreemFS) befindlichen Job Meta-Informationen und schickt

eine Antwort mit dem Status der Sicherung an die Anwendung. Nach dem Empfang der Antwort kann die Anwendung mit ihrer gewöhnlichen Ausführung fortfahren. Die Abbildung 3.5 stellt den Ablauf der Sicherung grafisch dar.

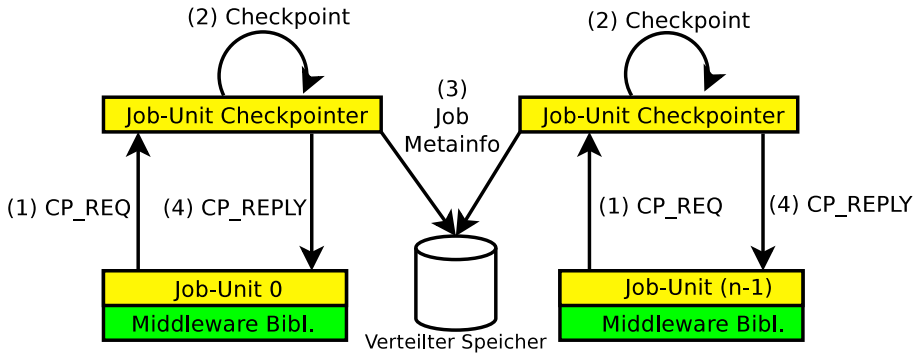

Abbildung 3.5: Durchführung einer unabhängigen Sicherung

Der Job-Unit Checkpointer wurde im Rahmen dieser Arbeit um die Funktionalität zum Annehmen der Checkpoint-Anfragen und zum Absenden der Antworten erweitert.

3.5 Wiederherstellung der verteilten Anwendung

Dieser Abschnitt beschäftigt sich mit der Wiederherstellung einer verteilten Anwendung. Dabei werden zunächst in Abschnitt 3.5.1 zwei Vorgehensweisen für die Wiederherstellung der Job-Units vorgestellt. Bei der ersten Vorgehensweise dürfen die zu wiederherstellenden Job-Units entweder als Client, oder als Server fungieren. Eine Kombination der beiden wird hierbei nicht unterstützt. Die zweite Vorgehensweise löst diese Einschränkung auf, indem es ein Koordinierungsprotokoll einsetzt, welches bei der Wiederherstellung aktiviert wird.
In den nachfolgenden zwei Abschnitten 3.5.2 und 3.5.3 werden zwei mögliche Wiederherstellungszenarien für Job-Units dargestellt. Bei beiden Szenarien ist entweder der Benutzer oder der XtreemGCP-Dienst selbst für die Initiierung einer Wiederherstellung verantwortlich.
Zunächst wird das erste Szenario erläutert. Bei diesem Szenario wird davon ausgegangen, dass die Wiederherstellung der verteilten Anwendung ausgehend von einem Totalausfall der Grid-Knoten stattfindet. Ein solches Szenario kann zwar unter Umständen eintreten, gilt jedoch in einer Grid-Umgebung mit einer hohen Anzahl von Knoten als sehr unwahr-

scheinlich.
Deswegen wurde im Rahmen dieser Arbeit eine weiteres Szenario für die Wiederherstellung einer verteilten Anwendung entwickelt. Bei diesem Szenario liegt nur ein Teilausfall der verteilten Anwendung und damit der Job-Units vor. Dies ist immer dann der Fall, wenn entweder ein Teil der Grid-Knoten, oder der Job-Units sich aufgrund von internen oder äußeren Ereignissen beenden sollten. In einem solchen Fall müssen zunächst die ausgefallenen Job-Units erkannt werden. Dafür wurde ein Ausfalls-Monitor entwickelt. Dieser Ausfalls-Monitor ist für die Aufzeichnung und Bereithaltung der Ausfallinformationen im Job-Checkpointer zuständig. Anschließend erfolgt die Berechnung eines global konsistenten Zustandes und die Wiederherstellung der Anwendung unter Berücksichtigung der Informationen, die von dem Ausfalls-Monitor bereitgestellt werden.

3.5.1 Vorgehensweisen bei der Wiederherstellung

Wie bereits in Abschnitt 3.1 erwähnt, unterstützt die im Rahmen dieser Arbeit implementierte Lösung für unabhängiges Checkpoint lediglich Job-Units, die entweder als Client, oder als Server fungieren. In der Realität operieren jedoch verteilte Anwendungen, sowohl als Client wie auch als Server. Dies hängt mit der Tatsache zusammen, dass für eine verteilte Berechnung in der Regel Daten zwischen den Job-Units in beide Richtigen ausgetauscht werden müssen. Dies tritt schon dann auf, wenn eine verteilte Berechnung stattfinden soll. In einem solchen Fall bekommt jede Job-Unit einen Teil der Daten und führt auf diesen Daten eine Operation aus. Am Ende werden die Teilberechnungen von einer zentralen Job-Unit zusammengefasst und formen dadurch ein Gesamtergebnis.
Die größte Herausforderung bei der Wiederherstellung einer solchen verteilten Anwendung besteht neben der Berechnung der Recovery-Line, in der Koordinierung der Job-Units während der eigentlichen Wiederherstellung. In Kontext von XtreemGCP ist der Job-Checkpointer für die Wiederherstellung der Job-Units zuständig. Dafür erkennt er beim koordinierten Checkpointing anhand von gespeicherten Meta-Informationen die Job-Units eines Jobs und stellt diese wieder her. Beim unabhängigen Checkpointing kommt zusätzlich noch die Erkennung der ausgefallenen Job-Units und die Berechnung der Recovery-Line anhand der zuvor aufgezeichneten Abhängigkeitsinformationen hinzu (siehe Abschnitt 3.5). Die eigentliche Wiederherstellung erfolgt dabei sowohl beim koordinierten, als auch beim unabhängigen Checkpointing parallel von den zuständigen Job-Unit Checkpointern. Da es sich bei den Job-Units um die Anwendungen handelt, die

in der Regel sowohl als Empfänger, wie auch als Sender fungieren können, kommt es an dieser Stelle zu einem zeitlichen Problem. Einige der Job-Units werden aufgrund der verwendeten Ressourcen (CPU, Speicher, etc.) langsamer als die anderen wiederhergestellt. Dadurch versuchen die zuerst wiederhergestellten Job-Units sich mit den noch nicht vorhanden Job-Units zu verbinden, um den Nachrichten durchzuführen. An dieser Stelle kommt es zu einem Konflikt, der aufgelöst werden muss.
Eine mögliche Lösung dieses Problems beruht auf der Annahme, dass die Job-Units nie gleichzeitig, sowohl als Client wie auch als Server fungieren können. Dadurch kann der Job-Checkpointer so modifiziert werden, dass zunächst die Clients und anschließend die Server wiederhergestellt werden. Die soeben erläuterte Problematik wird dadurch auf Kosten der Möglichkeit zum gegenseitigen Datenaustausch behoben. Dieser Ansatz wurde von Anfang an verfolgt und implementiert (dazu mehr in Kapitel 4).
Dennoch wurde im Verlauf dieser Arbeit ein alternativer Ansatz erarbeitet, der es erlaubt Job-Units wiederherzustellen, die sowohl als Client wie auch als Server fungieren können. Dafür berechnet der XtreemGCP zunächst einen global konsistenten Zustand und stellt die Job-Units wieder her. Anschließend sorgt eine von jeder Job-Unit dynamisch eingebundene Bibliothek durch das Abfangen des Systemaufrufs *connect()* dafür, dass keine Kommunikation mit den Job-Units stattfinden kann, die noch nicht wiederherstellt wurden. Dabei wird bei der Wiederherstellung der Job-Unit zunächst der initiale *connect()*-Aufruf abgefangen und eine Nachricht an den zuständigen Job-Unit Checkpointer geschickt (siehe Abbildung 3.6).

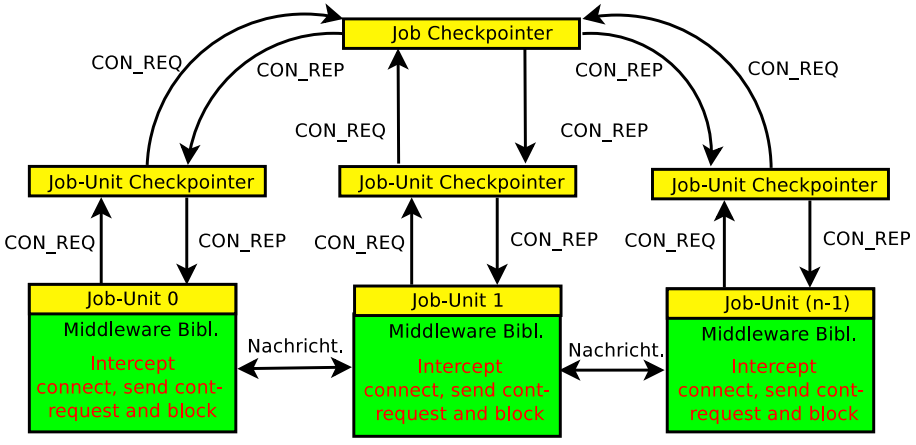

Abbildung 3.6: Wiederherstellung von Client-/Server Job-Units

Diese Nachricht enthält Informationen über den Empfänger (IP-Adresse, Port) und wird vom Job-Unit Checkpointer an den Job-Checkpointer weitergeleitet. Der Job-

Checkpointer erkennt anhand der übermittelten Informationen, ob der Empfänger bereits wiederhergestellt wurde und schickt im Erfolgsfall eine Nachricht über den Job-Unit Checkpointer an die zuständige Job-Unit zurück. Während der gesamten Zeit bleibt die Job-Unit blockiert und setzt erst mit dem Empfang einer positiven Nachricht ihre Arbeit fort. Dadurch wird stets sichergestellt, dass die Gegenseite vor der eigentlichen Kommunikation wiederhergestellt wurde.

3.5.2 Ausfall der gesamten Anwendung

Nachdem im vorherigen Abschnitt zwei Vorgehensweisen für die Wiederherstellung der Job-Units vorgestellt wurden, soll an dieser Stelle zunächst die Wiederherstellung der Anwendung im Falle eines Totalausfalls der Grid-Knoten ausführlich erläutert werden. In einem solchen Fall berechnet der Job-Checkpointer den global konsistenten Zustand eines Jobs, ausgehend vom letzten Checkpoint aller Job-Units und stellt anschließend die ausgefallenen Job-Units wieder her. Die nachfolgende Abbildung 3.7 stellt diesen Fall grafisch dar.

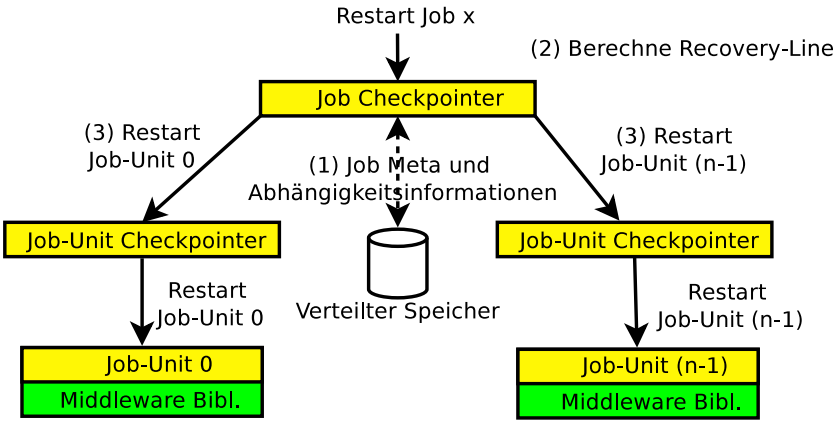

Abbildung 3.7: Verlauf der Wiederherstellung, Totalausfall

Zunächst ermittelt der Job-Checkpointer anhand der in Abschnitt 3.4 aufgezeichneten Job Meta-Informationen, die Job-Units des wiederherzustellenden Jobs. Anschließend werden die zuvor in Abschnitt 3.2 aufgezeichneten Abhängigkeitsinformationen der Job-Units zur Berechnung eines global konsistenten Zustandes der Anwendung verwendet. Die Berechnung erfolgt mittels des in Abschnitt 3.3 vorgestellten Rollback-Propagation Algorithmus. Zuletzt erfolgt die Wiederherstellung der Anwendung nach der in Abschnitt 3.5.1

erläuterten Vorgehensweise unter Beachtung der Client-/Server Reihenfolge. Dabei sorgt der Job-Checkpointer dafür, dass zunächst die Job-Units der Server und anschließend die der Clients mit Hilfe der entsprechenden Job-Unit Checkpointer wiederhergestellt werden.
Die nachfolgende Abbildung 3.8 stellt den soeben erläuterten Verlauf anhand einer Beispielanwendung, bestehend aus drei Job-Units grafisch dar. Dabei werden wie bereits in Abschnitt 2.2.1 vorgestellt, die Nachrichten durch Pfeile, und die Sicherungen durch blaue Punkte dargestellt. Die mittels der Berechnung der Recovery-Line für die Wiederherstellung ermittelten Sicherungen wurden rot markiert.

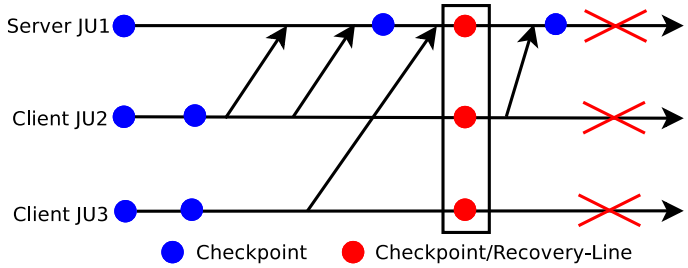

Abbildung 3.8: Beispielanwendung, Totalausfall

In dieser Abbildung wird ein Kommunikationsmuster der Beispielanwendung mit drei Job-Units *JU1*, *JU2* und *JU3* dargestellt, wobei sich alle Job-Units infolge von Grid-Knoten Ausfällen beendeten. Der Job-Unit Checkpointer erkennt die zu dem wiederherzustellenden Job gehörenden Job-Units anhand der zuvor aufgezeichneten Job Meta-Informationen, berechnet einen global konsistenten Zustand und stellt anschließend die ausgefallenen Job-Units unter Beachtung der Client-/Server Reihenfolge wieder her. Dadurch wird bei der soeben erläuterten Beispielanwendung zunächst die Server Job-Unit *JU1* und anschließend die Client Job-Units *JU2* und *JU3* wiederhergestellt.

3.5.3 Teilausfall der Anwendung

Im vorherigen Abschnitt wurde ein Szenario erläutert, in dem eine Wiederherstellung der verteilten Anwendung ausgehend von einem Ausfall aller Grid-Knoten durchgeführt wird. Da in einer produktiven Grid-Umgebung ein solcher Fall aufgrund der hohen Anzahl von Knoten unwahrscheinlich ist, wurde eine weitere Vorgehensweise entwickelt. Diese Vorgehensweise geht von einem Teilausfall der Anwendung aus und stellt lediglich die ausgefallenen Job-Units wieder her.

Für eine erfolgreiche Wiederherstellung der ausgefallenen Job-Units bedarf es zunächst eines Mechanismus zur Erkennung der Ausfälle. Dafür wurde der Job-Checkpointer um einen Ausfalls-Monitor erweitert. Der Ausfalls-Monitor kümmert sich um die Speicherung der Ausfallinformationen, die für die Wiederherstellung der Anwendung notwendig sind. Zunächst erkennt der ExecMng eines Grid-Knotens die Terminierung der Job-Units anhand derer Exit-Codes. Anschließend wird der zuständige Job-Checkpointer über einen Ausfall der Job-Unit benachrichtigt. Diese Information wird schließlich vom Job-Checkpointer in einem Ausfalls-Monitor festgehalten. Die oben genannte Ausfallerkennung wird in der Abbildung 3.9 grafisch dargestellt.

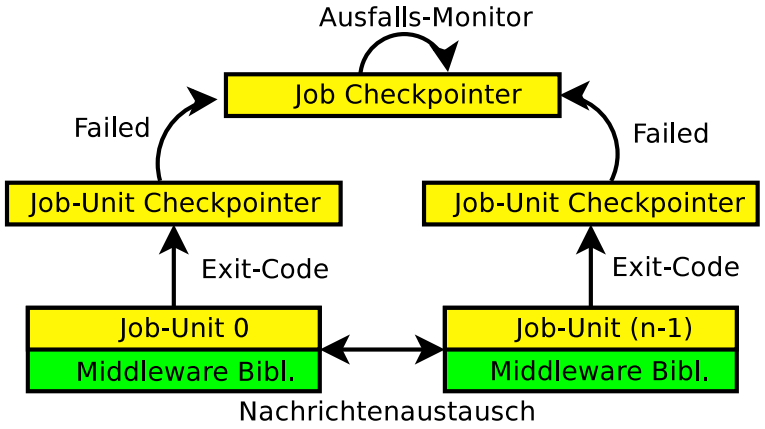

Abbildung 3.9: Funktionsweise des Ausfall-Monitors

Nach der Ausfallerkennung erfolgt die Wiederherstellung der ausgefallenen Job-Units. Dabei wird zunächst anhand der zuvor erstellten Job Meta-Informationen erkannt, welche Job-Units zu dem wiederherzustellenden Job gehören. Danach werden die ausgefallenen Job-Units anhand der Information aus dem Ausfalls-Monitor ermittelt. Anschließend erfolgt die Berechnung der Recovery-Line. Im Laufe der Berechnung werden zunächst die aufzeichneten Abhängigkeitsinformationen geladen. Anhand dieser Informationen wird der in Abschnitt 3.2 vorgestellte Checkpoint-Graph erzeugt. Auf diesen Graphen wird der in Abschnitt 3.3 erläuterte Rollback-Propagation Algorithmus zur Berechnung der Recovery-Line angewandt. Die aus der Berechnung resultierende Recovery-Line wird zuletzt vom Job-Checkpointer verwendet, um die Anwendung mit Hilfe der zuständigen Job-Unit Checkpointer in einen global konsistenten Zustand zu bringen. Dabei kann es durchaus vorkommen, dass einige der noch laufenden Job-Units aufgrund der ermittelten Abhängigkeiten zu einem früheren Zeitpunkt zurückgerollt werden müssen. Dafür bedient sich der Job-Checkpointer zunächst der Information aus dem Ausfalls-Monitor, um den Status der noch laufenden Job-Units zu ermitteln. Falls die Job-Units noch laufen sollten,

werden sie vom Job-Checkpointer mit Hilfe des zuständigen Job-Unit Checkpointers beendet. Nach der Beendigung der Job-Units erfolgt die Wiederherstellung des global konsistenten Zustands. Im Rahmen dieses Vorgangs, weist der Job-Checkpointer die Job-Unit Checkpointer an, die Job-Units zu einer vorgegebenen Version, unter Beachtung der Client-/Server Reihenfolge zurückzurollen. Zuletzt muss die Information über den Status der wiederhergestellten Job-Units dem Job-Checkpointer und dessen Ausfalls-Monitor mitgeteilt werden. Dafür sendet der jeweilige Job-Unit Checkpointer nach einer erfolgreichen Wiederherstellung, eine Aktualisierungsnachricht an den Job-Checkpointer. Der in diesem Abschnitt erläuterte Zusammenhang wird in der Abbildung 3.10 grafisch dargestellt.

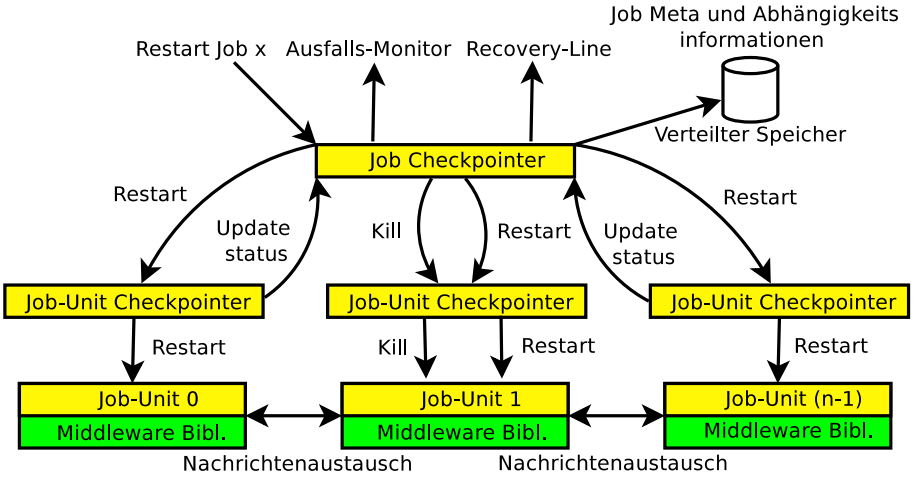

Abbildung 3.10: Verlauf der Wiederherstellung, Teilausfall

Wie soeben erläutert, kann es durchaus vorkommen, dass einige der noch laufenden Job-Units aufgrund der ermittelten Abhängigkeiten zu einem früheren Zeitpunkt zurückgerollt werden müssen. Der Ablauf einer solchen Wiederherstellung soll anhand dem nachfolgenden Beispielanwendung, bestehend aus drei Job-Units verdeutlicht werden.

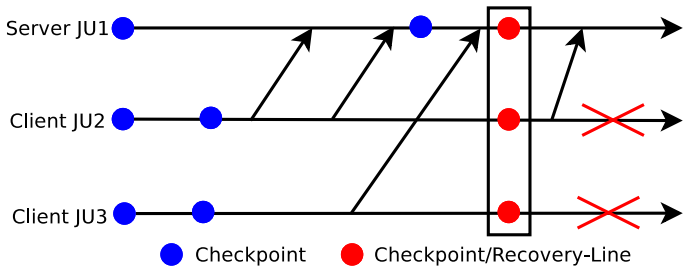

Abbildung 3.11: Beispielanwendung, Teilausfall

Wie der Abbildung 3.11 zu entnehmen, fallen bei dieser Anwendung zwei Job-Units *JU2* und *JU3* aus, wobei die erste Job-Unit *JU1* erhalten bleibt. Aufgrund der berechneten Recovery-Line, muss die erste noch laufende Job-Unit zu einem früheren Zeitpunkt zurückgerollt werden. Dafür erkennt der Job-Checkpointer zunächst anhand der Information aus dem Ausfalls-Monitor, dass die *JU1* noch läuft und beendet sie. Anschließend stellt der Job-Checkpointer die Job-Units mit Hilfe der zuständigen Job-Unit Checkpointer, entsprechend der von der Recovery-Line vorgegebene Version, wieder her und aktualisiert die Informationen im Ausfalls-Monitor.

Im nachfolgenden Kapitel 4 wird die Implementierung des im Rahmen dieses Kapitels vorgestellten konzeptuellen Entwurfs ausführlich erläutert.

Kapitel 4

Implementierung des unabhängigen Checkpointings

In diesem Kapitel wird die Implementierung des im vorherigen Kapitel vorgestellten Entwurfs zum unabhängigen Sichern und Wiederherstellen der verteilten Anwendungen unter XtreemGCP vorgestellt.
Die Implementierung besteht aus vier Hauptkomponenten, die ein unabhängiges Sicherungsverfahren den Anwendungen zur Verfügung stellen. Sie kümmern sich um die Ausfallerkennung, Recovery-Line Berechnung, Wiederherstellung der ausgefallenen Anwendungsteile und einen adaptiven Wechsel der Checkpoint-Strategie.
Die erste Komponente, genannt CRUC (Checkpoint/Restart Uncoordinated), ist eine dynamische Bibliothek, die zur Verwaltung und Aufzeichnung der Inter-Prozeß-Abhängigkeiten innerhalb der verteilten Anwendung eingesetzt wird. Ihre Aufgabe besteht darin, die Systemaufrufe *send()* und *recv()* abzufangen, um den ausgehenden Nachrichten die Abhängigkeitsinformationen hinzuzufügen und die Informationen aus den eingehenden Nachrichten zu extrahieren. Diese Informationen werden auf einem persistenten Speicher (z.B. XtreemFS) für die spätere Berechnung eines global konsistenten Zustandes festgehalten. Zusätzlich stellt die CRUC-Bibliothek dem Anwendungsentwickler eine Schnittstelle zur Verfügung, um die Sicherungen zu einem beliebigen Zeitpunkt vorzunehmen.
Die zweite Komponente ist eine weitere dynamische Bibliothek, genannt CRUC-API. Sie stellt eine Schnittstelle für die Kommunikation mit der CRUC-Bibliothek zur Verfügung. Die CRUC-API Bibliothek kann für die Erstellung der unabhängigen Sicherung von allen Teilen einer verteilten Anwendung eingebunden werden. Damit entfällt die

Notwendigkeit einer statischen Einbindung der kompletten CRUC-Bibliothek.
Die dritte Komponente besitzt keinen eigenständigen Namen und befindet sich im Gegensatz zu allen anderen Komponenten innerhalb des Grid-Checkpointers. Sie wird dafür eingesetzt, Checkpoint-Anfragen der CRUC-Bibliothek an die entsprechenden Checkpointer zur Sicherung der Anwendung weiterzuleiten. Außerdem ist sie für die Aktualisierung der Job Meta-Informationen und die Berechnung der Recovery-Line während der Wiederherstellung verantwortlich. Mit Hilfe der aktualisierten Job Meta-Informationen und des Rollback-Propagation Algorithmus ist diese Komponente in der Lage ausgefallene Anwendungskomponenten konsistent wiederherzustellen.
Bei der vierten Komponente, genannt *LibSwitch*, handelt es sich um eine Bibliothek zum adaptiven Wechsel der Checkpoint-Strategie (Koordiniert/Unkoordiniert). Sie ist für das Abfangen von Systemaufrufen und deren Weiterleitung an die dynamisch ausgewählte Checkpoint-Bibliothek zuständig.

4.1 Abfangen von Bibliotheks- oder Systemaufrufen

Die CRUC-Komponente fängt alle Sende- und Empfangs-Ereignisse der Anwendung ab. Anschließend werden diese Nachrichten analysiert und verändert. Im Falle eines Sende-Vorgangs müssen der ausgehenden Nachricht die Abhängigkeitsinformationen, bestehend aus der PID des Prozesses und einem Checkpoint-Index, hinzugefügt werden (siehe Abschnitt 3.2). Bei einem Empfangsereignis werden die Abhängigkeitsinformationen aus der Nachricht extrahiert. Diese Schritte erfordern einen Mechanismus, der das Abfangen und Manipulieren der entsprechenden Bibliotheks- oder Systemaufrufe erlaubt. Dafür existieren zwei Vorgehensweisen mit jeweils unterschiedlichen Vor- und Nachteilen.
Das erste Verfahren ist Bibliotheks-basiert und beruht auf dem Abfangen der Bibliotheks-Aufrufe. Linux basierte Betriebssysteme bieten mit der GLIBC eine Bibliothek an, die einen Wrapper für die eigentlichen Systemaufrufe darstellt. Diese Wrapper-Methoden lassen sich in einer eigenen Bibliothek implementieren und können mit Hilfe des LD_PRELOAD-Mechanismus die nativen Methoden der GLIBC [Gnu07] überschreiben. Dadurch werden alle Aufrufe an die neue Bibliothek geleitet und können von dort aus erweitert werden, ohne die nativen GLIBC-Aufrufe verändern zu müssen. Der Vorteil dieses Verfahrens liegt in der leichten Implementierbarkeit und in der Effizienz. Da die Systemaufrufe mittels der Software-Interrupts direkt getätigt werden

können, stellt dies einen Nachteil des Verfahrens dar. In diesem Fall werden die GLIBC Wrapper-Methoden komplett umgangen.[JS00]. Damit ist dieser Ansatz nur teilweise zum Abfangen der Systemaufrufe geeignet.
Ein alternatives Verfahren fängt die Systemaufrufe direkt auf der Kernel-Ebene ab, indem es die Systemaufruf-Tabelle mit Hilfe eines speziellen Kernel-Moduls modifiziert und mit den eigenen Aufrufen ersetzt. Da eine solche Änderung das gesamte System betrifft, werden dabei die Systemaufrufe aller Prozesse abgefangen. Ein solches Verhalten kann zwar z.B. zur Erstellung von Zugriffsbeschränkungen für bestimmte Prozesse genutzt werden, ist jedoch für diese Arbeit weniger relevant. Kerneländerungen stellen ein Sicherheitsrisiko dar, sind stark versionsabhängig und beeinflussen das gesamte System. Außerdem dürfen die Kernelmodule nur von den Administratoren des Systems geladen werden. Damit schränken sie die Einsatzfähigkeit der Software erheblich ein.
Aufgrund der einfacheren und effizienteren Implementierungsmöglichkeiten, wird im Rahmen dieser Arbeit das zuerst beschriebene Verfahren verwendet.

4.2 Aufbau der CRUC-Bibliothek

In diesem Abschnitt wird der Aufbau der CRUC-Bibliothek erläutert. Dafür stellt zunächst die Abbildung 4.1 die Struktur der CRUC-Bibliothek dar.

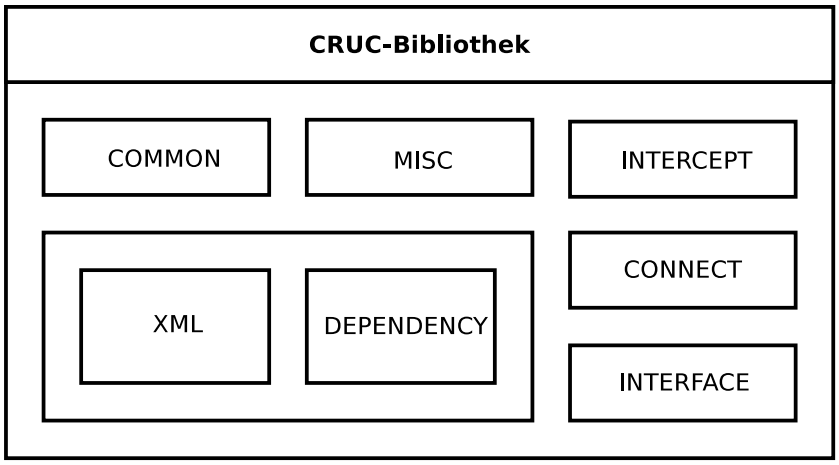

Abbildung 4.1: Aufbau der CRUC-Bibliothek

Die Aufgaben der einzelnen Teilkomponenten der Bibliothek werden in der nachfolgenden Tabelle 4.1 beschrieben.

Name	Erläuterung
Common-Modul	Automatische Konfiguration der Bibliothek. Stellt derzeit einen Parser für Konfigurationsdateien bereit
Misc-Modul	Beinhaltet diverse Hilfsfunktionen
Intercept-Modul	Übernimmt neben dem Abfangen der Systemaufrufe auch das Anhängen und Extrahieren der Abhängigkeitsinformationen eines Prozesses
XML-Modul	Speicherung der Abhängigkeitsinformationen
Dependency-Modul	Verwaltung der Abhängigkeiten
Connect-Modul	Stellt die Routinen zur Herstellung der Verbindung mit der Grid-Ebene zur Verfügung
Interface-Modul	Stellt eine Schnittstelle zur Kommunikation mit der CRUC-Bibliothek bereit

Tabelle 4.1: Komponenten der CRUC-Bibliothek

4.2.1 Automatische Konfiguration der Bibliothek

Da die CRUC-Bibliothek sowohl für die Aufzeichnung der Abhängigkeiten, als auch für die Benachrichtigung des Grid-Checkpointers zuständig ist, hängt sie von einigen, wichtigen Parametern ab. Diese Angaben betreffen derzeit den Dateinamen und das Verzeichnis zum Ablegen der Abhängigkeitsinformationen. Darüber hinaus müssen der Bibliothek die Kontaktinformationen des Servers auf der Grid-Ebene mitgeteilt werden. Somit kann der Server im Falle eines Checkpoints benachrichtigt werden. All diese Angaben müssen während der Initialisierungsphase gesetzt werden und gelten anschließend für die gesamte Laufzeit der Anwendung. Diese Informationen werden mittels einer Konfigurationsdatei bereitgestellt. Der dafür entwickelte Konfigurations-Parser ist ein Bestandteil des in Abschnitt 4.2 vorgestellten Common-Moduls.
Zunächst wird während der Bibliothekseinrichtung, in dem Verzeichnis */etc/xos/cruc* eine Konfigurationsdatei namens *„xos_cruc.conf“* automatisch erstellt. Der Inhalt dieser Datei wird in der Auflistung 4.1 dargestellt und setzt sich aus dem Abschnitt *data_settings* und *host_settings* zusammen.

Listing 4.1: Aufbau der Konfigurationsdatei (xos_cruc.conf)

```
[data_settings]
data.dir = /xtreemfs/cruc
data.file = dep_list
[host_settings]
listen.address = 127.0.0.1
listen.port = 5000
```

Der erste Abschnitt *data_settings* beinhaltet das Verzeichnis (data.dir) und den Präfix des Dateinamens (data.file) zur Speicherung der Abhängigkeitsinformationen für den jeweiligen Prozess. Im zweiten Abschnitt *host_settings* werden die Informationen für die Verbindung mit dem Servers auf der Grid-Ebene, festgelegt. An dieser Stelle muss sowohl die Adresse (listen.address) als auch die Port-Nummer (listen.port) des Servers angegeben werden. Diese Verbindungsinformationen werden von der CRUC-API verwendet, um eine Verbindung zum XtreemGCP herzustellen und eine Anforderung zum unabhängigen Checkpoint der Anwendung auszulösen. In den meisten Fällen reicht es hierbei die IP-Adresse des lokalen Rechners anzugeben. Eine Änderung dieser Einstellung muss erst dann erfolgen, wenn der Empfangsserver innerhalb von XtreemGCP an eine andere Adresse gebunden wurde.
Die in der Auflistung 4.2 dargestellte Struktur *struct config_params* repräsentiert den Inhalt einer Konfigurationsdatei.

Listing 4.2: Struktur zur Repräsentation der Konfigurationsdatei

```
struct config_params {
      char *data_dir;
      char *data_file;
      char *listenaddr;
      int listenport;
};
```

Die vom Konfigurationsdatei-Parser exportierten Funktionen werden in der Auflistung 4.3 dargestellt.

Listing 4.3: Funktionen des Parsers

```
struct config_params *cruc_parse_config(void);
struct config_params *cruc_get_config(void);
```

Die Funktion *cruc_parse_config()* liest den Inhalt der Konfigurationsdatei ein und liefert die ausgelesenen Daten in Form einer Struktur. Die zurückgegebene Struktur wird während der gesamten Laufzeit der Anwendung im Speicher für weitere Aufrufe gehalten und kann mit Hilfe der *cruc_get_config()*-Funktion abgefragt werden.

4.2.2 Hilfsfunktionen der Bibliothek

Das Misc-Modul der CRUC-Bibliothek beinhaltet die Hilfsfunktionen, die überwiegend für interne Verwaltungszwecke verwendet werden. Die folgenden, in der Auflistung 4.4 dargestellten Routinen, werden zur Verwaltung des in Kapitel 3 eingeführten Checkpoint-Indexes eingesetzt.

Listing 4.4: Verwaltung vom Checkpoint-Index

```
1 void cruc_inc_checkpoint_index(void);
2 int64_t cruc_get_checkpoint_index(void);
```

Mittels der *cruc_inc_checkpoint_index()*-Funktion wird der interne, in Kapitel 3 beschriebene Checkpoint-Index inkrementiert.
Da die für die Wiederherstellung der Anwendung notwendigen Abhängigkeitsinformationen auf einem persistenten Speicher gespeichert werden müssen, führt die Misc-Komponente die in der Auflistung 4.5 dargestellte Aufzählung *cruc_file* ein, in der die erlaubten Dateitypen definiert werden. Der Typ *CURRENT* legt den Namen der Datei fest, in der die Abhängigkeitsinformationen der noch laufenden Anwendungen gesichert werden. Der Dateityp *VERSION* wird für die Erzeugung einer Datei während des Checkpoint-Vorgangs verwendet. Die Dateien vom Typ *VERSION* beinhalten somit die zu einem Checkpoint zugehörigen Abhängigkeitsinformationen.

Listing 4.5: Festlegung des Dateinamens

```
1 typedef enum cruc_file {
2       CURRENT,
3       VERSION,
4 } cruc_file_t;
```

Die Auflistung 4.6 stellt die Funktionen für die Ordner- und Dateioperationen dar.

Listing 4.6: Verwaltung der Dateien

```
1 char *cruc_get_filename(cruc_file_t filetag);
2 int cruc_create_directory(void);
```

Die Funktion *cruc_get_filename()* wird für die Generierung des Namens der Datei zum Aufbewahren der Abhängigkeiten eingesetzt. Sie bekommt als Parameter die Auflistung *cruc_file_t* und liefert als Rückgabeparameter den entsprechenden Dateinamen *dep_list_current.xml* bzw. *dep_list_vX.xml* zurück.
Alle Abhängigkeitsinformationen der verteilten Anwendung werden in dem im Abschnitt 4.2.1 definierten Verzeichnis (*/xtreemos/cruc/*) gespeichert. Dabei wird für jeden Prozess ein Unterverzeichnis für die Abhängigkeitsinformation-Dateien erstellt. Die Erstellung dieser Verzeichnisse wird von der *cruc_create_directory()*-Routine übernommen. Die Routine liest dabei die PID des aktuellen Prozesses aus. Mit Hilfe dieser ID wird ein Unterverzeichnis (*/xtreemos/cruc/PID*) erstellt.
Wie bereits in Kapitel 3 erläutert, ist die derzeitige Implementierung auf verteilte Anwendungen beschränkt, die ausschließlich aus einem Server und mehreren Clients bestehen dürfen. Bei einem solchen Anwendungstyp müssen im Fehlerfall zuerst die Server und anschließend die Clients wiederhergestellt werden. Das im nachfolgenden Abschnitt 4.2.3 näher erläuterte Intercept-Modul erkennt, ob es sich bei dem Prozess um einen Client oder einen Server handelt und speichert diese Informationen mit Hilfe der in Auflistung dargestellten Funktionen 4.7 ab.

Listing 4.7: Verwaltung der Server-Information

```
1 void cruc_set_server(void);
2 int cruc_get_server(void);
```

4.2.3 Abfangen der Systemaufrufe und Kapselung der Abhängigkeitsinformationen

Für die Aufzeichnung der Abhängigkeiten und die Ermittlung, ob es sich bei dem Prozess um einen Server oder Client handelt, werden derzeit die folgenden Systemaufrufe von der Intercept-Komponente abgefangen: *send()*, *recv()* und *bind()*. Nach dem Abfangen des Sende-Aufrufes werden die ausgehenden Nachrichten manipuliert und mit Abhängigkeitsinformationen versehen (siehe Abschnitt 4.1). Auf der Empfänger-Seite wird der

Empfangs-Aufruf abgefangen, um die hinzugefügten Informationen zu extrahieren. Mit Hilfe des *bind*-Aufrufes wird außerdem festgestellt, ob es sich bei dem Prozess um einen Server handelt. Dies ist für die spätere Wiederherstellung der Anwendung von großer Bedeutung.
Das Abfangen der Systemaufrufe geschieht dabei entsprechend der im Abschnitt 4.1 erläuterten LD_PRELOAD-Methode. Für die Übertragung der Abhängigkeitsinformationen kamen zwei Ansätze in Frage:

1. Die Abhängigkeitsinformationen können an die eigentliche Nachricht angehängt werden. Bei näherer Betrachtung dieses Ansatzes ergeben sich negative Auswirkungen auf die Performance des Systems. Zum Anhängen der Zusatzinformationen an eine Nachricht, muss der übergebene Nachrichtenpuffer umkopiert werden. In Anbetracht dieser Tatsache stellt sich die Vorgehensweise für diese Arbeit als nicht relevant heraus.

2. Sende-Ereignis kann folgendermaßen aufgeteilt werden. Im ersten Schritt wird die erstellte Abhängigkeitsinformation übertragen. Im zweiten Schritt folgt schließlich die Übertragung der eigentlichen Nachricht. Dafür werden alle Sende-Ereignisse mit Hilfe einer Wrapper-Methode abgefangen und wie in der Auflistung 4.8 verkürzt dargestellt, modifiziert.

   Listing 4.8: Absenden der Abhängigkeitsinformationen

```
1 ssize_t send(int __fd, __const void *__buf, size_t __n, int __flags) {
2       if (__send == NULL) __send = dlsym(RTLD_NEXT, "send");
3       dentry.pid = getpid();
4       dentry.cp_idx = cruc_get_checkpoint_index();
5       __send(__fd, &dentry, sizeof(struct dep_entry_cnt), __flags);
6       return __send(__fd, __buf, __n, __flags);
7 }
```

   Beim ersten Sende-Aufruf wird mit Hilfe des *dlsym*-Aufrufs die Referenz auf das native send() der GLIBC geholt und in einer globalen Variable festgehalten. Anschließend wird die in der Auflistung 4.9 dargestellte Struktur *struct dep_entry_cnt* zur Repräsentation der Abhängigkeitsinformation angelegt und mit der aktuellen PID und dem aktuellen Checkpoint-Index gefüllt. Mittels der drauf folgenden *__send()*-Aufrufe wird die Abhängigkeitsinformation und die ursprüngliche Nachricht übertragen.

Listing 4.9: Struktur für Abhängigkeitsinformationen

```
1 struct dep_entry_cnt {
2       pid_t pid;
3       int64_t cp_idx;
4 };
```

Bei dem Empfänger verhält sich der Empfangs-Aufruf analog zu dem Sende-Aufruf. Die Abhängigkeitsinformationen werden zuerst übertragen. Anschließend wird die originelle Nachricht empfangen. Bei einer derartigen Übertragungsmethode müssen die Nachrichten weder umkopiert noch manipuliert werden. Dadurch wird der Speicherverbrauch vermindert.

Nach dem Empfang einer Nachricht muss die entsprechende Abhängigkeitsinformation der Dependency-Komponente übergeben werden. Dafür wird mittels des Empfang-Wrappers die von der Dependency-Komponente zur Verfügung gestellte Funktion *cruc_update_depend_table()* aufgerufen (siehe Abschnitt 4.2.4). Sie sorgt schließlich für die ordnungsgemäße Aktualisierung und Speicherung der Abhängigkeitsinformationen. Intercept-Modul erkennt, ob es sich bei dem Prozess um einen Client oder einen Server handelt. Für die Umsetzung dieser Funktionalität, fängt das Intercept-Modul den Aufruf der GLIBC Funktion *bind()* ab und ermittelt somit, ob es sich hierbei um einen Server- oder Client-Prozess handelt. Die Auflistung 4.10 stellt diesen Vorgang verkürzt dar.

Listing 4.10: Client-Server Erkennung

```
1 int bind (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t __len) {
2       if (__bind == NULL) __bind = dlsym(RTLD_NEXT, "bind");
3       cruc_set_server();
4       return __bind(__fd, __addr, __len);
5 }
```

Dabei wird der *bind*-Aufruf abgefangen und eine von der Misc-Komponente zur Verfügung gestellte Routine *cruc_set_server()* aufgerufen. Diese Information wird zu einem späteren Zeitpunkt mittels der Checkpoint-Anfrage an den XtreemGCP übergeben (siehe Abschnitt 4.3) und auf der Grid-Ebene für die spätere Wiederherstellung festgehalten.

4.2.4 Verwaltung der Abhängigkeitsinformationen

Bei der Verwaltung der Abhängigkeitsinformationen spielen zwei Komponenten eine wichtige Rolle. Die XML-Komponente ist für die Speicherung der Abhängigkeitsinformationen zuständig. Ihre Aufgabe besteht darin, die Abhängigkeitsinformationen auf einem persistenten Speicher in Form einer XML-Datei festzuhalten. Die nachfolgende Auflistung 4.11 stellt die von XML-Komponente zur Verfügung gestellte Schnittstellen dar.

Listing 4.11: Verwaltung der XML-Datei

```
1 xmlTextWriterPtr cruc_init_dep_file(const char *file);
2 int cruc_create_dep_entry(xmlTextWriterPtr xml_writer, struct dep_entry_cnt *
      dentry);
3 int cruc_fini_file(xmlTextWriterPtr xml_writer);
```

Die Routine *cruc_init_dep_file* dient der Erstellung einer XML-Datei. Als Parameter wird der Dateinamen übergeben, als Rückgabewert gibt die Funktion einen XMLWriter-Zeiger *xmlTextWriterPtr* zurück. Dieser Zeiger wird von der *cruc_create_dep_entry()*-Funktion für die Erstellung neuer Dateieinträge verwendet. Jeder Eintrag wird innerhalb der in Abschnitt 4.2.3 dargestellten Struktur *struct dep_entry_cnt* abgebildet.
Um die Aufzeichnung der Abhängigkeitsinformationen kümmert sich die Dependency-Komponente. Die Informationen werden in Form einer XML-Datei gespeichert. Für die Verwaltung der Abhängigkeiten benutzt die CRUC-Bibliothek eine von der GLIBC bereitgestellte Implementierung der Hashtabellen. Darin werden die empfangenen Prozess-Abhängigkeiten festgehalten. Die PID des Absenders fungiert dabei als der eindeutige Schlüssel. Jeder Eintrag der Hashtabelle wird durch eine im Abschnitt 4.2.3 eingeführte Struktur *struct dep_entry_cnt* repräsentiert.
An dieser Stelle ist es wichtig zu erwähnen, dass von der Eindeutigkeit der PID des Absenders im Grid nicht immer ausgegangen werden kann. Es ist durchaus möglich, dass zwei Sender-Prozesse, die auf unterschiedlichen Grid-Knoten ausgeführt werden, die gleiche PID erhalten. In einem solchen Fall kann es auf der Empfängerseite zu Kollisionen kommen. Dieser Fall wird von der derzeitigen Implementierung nicht abgedeckt und bedarf der Einführung einer Struktur zur eindeutigen Identifizierung von Prozessen innerhalb eines Grids .
Die in der Auflistung 4.12 dargestellten Funktionen sind für das Anlegen der Verwaltungsstruktur und für das Löschen aller Elemente zuständig.

Listing 4.12: Initialisierung der Verwaltungsstruktur

```
1 int cruc_init_depend_table (void);
2 void cruc_free_depend_table (void);
```

Die Funktion *cruc_init_depend_table()* ist für die Initialisierung der Verwaltungsstruktur zuständig. Somit wird sie bei der ersten Sicherung der Anwendung ausgeführt. Außerdem muss nach jedem getätigten Checkpoint-Vorgang die Aufzeichnung der Abhängigkeiten für den nachfolgenden Checkpoint-Abschnitt erneut erfolgen (siehe Abschnitt 3). Dafür werden alle Einträge der aktuellen Verwaltungsstruktur mittels des Funktionsaufrufs *cruc_free_depend_table()* gelöscht.
Neben den oben genannten Funktionen für die Initialisierung und Freigabe der Verwaltungsstruktur, exportiert das Dependency-Modul die in der Auflistung 4.13 dargestellten Routinen für die Verwaltung der Abhängigkeitsinformationen.

Listing 4.13: Operationen auf der Verwaltungsstruktur

```
1 int cruc_init_current_info ();
2 int cruc_update_depend_table (struct dep_entry_cnt *cont);
3 int cruc_save_depend_table (cruc_file_t filetag);
4 int cruc_print_depend_table (void);
```

Wie bereits in Kapitel 3 erwähnt, müssen Abhängigkeitsinformationen sowohl zum Checkpoint-Zeitpunkt als auch während der Laufzeit einer Anwendung festgehalten werden. Für diesen Vorgang stellt die Dependency-Komponente die Funktion *cruc_init_current_info()* bereit. Sie wird bei jedem Checkpoint-Aufruf aufgerufen und sorgt dafür, dass eine Abhängigkeit zum vorherigen Checkpoint des gleichen Prozesses in die Verwaltungsstruktur eingetragen und auf der Festplatte in Form einer XML-Datei (dep_list_current.xml) gespeichert wird (siehe Abbildung 4.2).

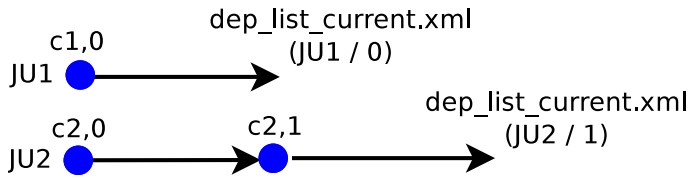

Abbildung 4.2: Anlegen der Current-Datei

Für die Aktualisierung der Verwaltungsstruktur, ist die *cruc_update_depend_table()*-Routine zuständig. Sie wird vom Intercept-Modul bei jeder eingehenden Abhängig-

keit aufgerufen und sorgt dafür, dass die Abhängigkeitsinformation in die Verwaltungsstruktur eingetragen wird. Außerdem ist sie sich für die Aktualisierung der XML-Datei (dep_list_current.xml) zuständig.
Da die Abhängigkeitsinformationen sowohl zur Laufzeit einer Anwendung als auch während des Checkpoints auf der Festplatte gespeichert werden müssen, wurde eine generische Routine *cruc_save_depend_table()* zur Speicherung dieser Informationen definiert. Sie bekommt als Parameter den im Abschnitt 4.2.2 definierten Dateibezeichner *cruc_file_t* und speichert die Abhängigkeitsinformationen unter diesem Bezeichner auf der Festplatte. Falls der Eintrag *VERSION* gesetzt werden sollte, werden die Abhängigkeitsinformationen unter dem Namen *dep_list_vX.xml* gespeichert. Andernfalls erfolgt die Speicherung unter dem Namen *dep_list_current.xml*. Diese Routine wird sowohl für die Aktualisierung der Current-Datei als auch von für die Abspeicherung der Abhängigkeiten zum Checkpoint-Zeitpunkt eingesetzt.
Die Routine *cruc_print_depend_table ()* gibt den Inhalt der aktuellen Verwaltungsstruktur aus. Diese Informationen werden zum Debuggen der Anwendung verwendet.

4.2.5 Schnittstelle zur CRUC-Bibliothek

Die oben genannten Funktionen sind vorwiegend für die Verarbeitung der Abhängigkeitsinformationen zuständig. Die nachfolgende Komponente ist für die Kommunikation mit dem Grid-Checkpointer erforderlich. Sie stellt dem Programmierer eine Schnittstelle zur Verfügung, um Checkpoint-Anfragen zu einem beliebigen Zeitpunkt aus einer Anwendung heraus auszulösen. Diese Komponente wird innerhalb des Interface-Moduls implementiert und kann entweder direkt, mittels der statischen Verlinkung der CRUC-Bibliothek mit der Anwendung oder dynamisch, mittels der im nachfolgenden Abschnitt 4.3 erläuterten CRUC-API, verwendet werden. Die von der CRUC-Bibliothek zur Verfügung gestellte Schnittstelle wird in der Auflistung 4.14 dargestellt.

Listing 4.14: Schnittstelle der CRUC-Bibliothek

```
1 int cruc_trigger_independent(struct cp_info *info);
2 int cruc_trigger_callback(void);
```

Der erste Aufruf *cruc_trigger_independent()* ist für das Auslösen einer Sicherung zuständig. Zunächst wird der Inhalt der in Abschnitt 4.2.1 vorgestellten Konfigurationsdatei

mit Hilfe der *cruc_get_config()*-Routine ausgelesen. Anschließend wird anhand dieser Information eine Verbindung mit dem Grid-Checkpointer aufgebaut und eine auf der übergebenen Struktur *struct cp_info* basierende Anfrage an den Grid-Checkpointer gesendet. Der Grid-Checkpointer verarbeitet diese Anfrage und erstellt daraufhin eine Anwendungssicherung. Abschließend wird eine Antwort über den Erfolg der Sicherung an die Bibliothek zurückgesendet. Die in der Auflistung 4.15 dargestellte *cr_info* Struktur wird für die Festlegung der von dem Grid-Checkpointer auszuführenden Anweisung verwendet.

Listing 4.15: Struktur für die Initialisierung

```
1 struct cp_info {
2       enum cp_strategy strategy;
3       enum cp_options options;
4       char jsdl_path[CRAPI_JSDL_PATH_SIZE];
5       int job_unit_id;
6 };
```

Der erste Parameter *strategy* definiert die Checkpoint-Strategie. Als Checkpoint-Strategie kann koordiniertes oder unabhängiges Checkpointing ausgewählt werden. Der zweite Parameter *options* legt fest, ob die Sicherung sequenziell oder inkrementell erfolgen soll. Zusätzlich muss derzeit aufgrund von Einschränkungen der AEM, der vollständige Pfad zu der Job-Unit JSDL-Datei angegeben werden. Diese Datei beinhaltet neben dem Pfad der ausführbaren Datei der Anwendung, die Angaben zu den benötigten Ressourcen. Mit Hilfe der JSDL-Datei ist der XtreemGCP in Lage, die Job-Meta Informationen während der unabhängigen Sicherung ordnungsgemäß zu aktualisieren. Eine weitere Einschränkung betrifft die Job-Unit IDs eines Jobs. Derzeit sieht die AEM keinerlei solcher IDs vor. Als eine vorübergehende Lösung wird deswegen die Job-Unit ID der Anwendungen erzeugt und dem Grid-Checkpointer während des unabhängigen Checkpointings übergeben. Diese ID wird für die Aktualisierung der Job-Meta Informationen benötigt und muss derzeit vom Anwendungsentwickler so gewählt werden, dass möglichst keine Kollisionen mit den IDs anderer Job-Units entstehen können.
Die Funktion *cruc_trigger_callback* muss während des Sicherungsvorgangs vom Anwendungsentwickler, aus einem Checkpoint-Callback heraus aufgerufen werden. Dadurch wird der Checkpoint-Index inkrementiert. Außerdem werden die Abhängigkeitsinformationen zur Berechnung der Recovery-Line in Form einer XML-Datei aufgezeichnet.

4.3 Dynamisches Laden der CRUC-Bibliothek

Die im vorherigen Abschnitt 4.2 vorgestellte CRUC-Bibliothek kann folgendermaßen eingebunden und benutzt werden. Die erste Methode wurde bereits mehrfach erwähnt und setzt eine statische Verlinkung der CRUC-Bibliothek voraus. Bei der zweiten Methode werden die Routinen der CRUC-Bibliothek dynamisch aufgerufen. Zu diesem Zweck wurde eine weitere Bibliothek, CRUC-API entwickelt. Sie muss statisch gegen die Anwendung gelinkt werden und beinhaltet lediglich die für die Erstellung einer Sicherung notwendigen Routinen. Ihre Aufgabe besteht darin die CRUC-Bibliothek dynamisch zu laden und deren in Abschnitt 4.2.5 vorgestellte Schnittstelle anzusprechen.
Dafür stellt die CRUC-API Bibliothek dem Anwendungsentwickler die in der nachfolgenden Auflistung 4.16 dargestellte Schnittstelle zur Verfügung.

Listing 4.16: Schnittstelle der CRUC-API Bibliothek

```
1 int cruc_checkpoint(struct cp_info *info);
2 int cruc_checkpoint_callback(void);
```

Die erste Funktion *cruc_checkpoint()* ist ein blockierender Aufruf zur Erstellung der Anwendungssicherung. Für die Nutzung dieser Schnittstelle muss die in Abschnitt 4.2.5 erläuterte Struktur *struct cr_info* initialisiert und als Parameter übergeben werden. In Folge dessen wird die CRUC-Bibliothek dynamisch geladen und die dafür vorgesehene Funktion *cruc_trigger_independent()* aufgerufen.
Die zweite Funktion *cruc_checkpoint_callback()* muss vom Anwendungsentwickler, aus einem Checkpoint-Callback heraus aufgerufen werden. Mit Hilfe dieses Aufrufs wird analog zu dem Aufruf der ersten Funktion, die CRUC-Bibliothek bei Bedarf geladen. Anschließend wird die dafür vorgesehene Schnittstelle *cruc_trigger_callback()* ausgeführt.

4.4 Erweiterungen des Grid-Checkpointers

Dieser Abschnitt behandelt die Erweiterungen der XtreemOS Grid-Checkpointer Komponenten. Im Rahmen des unabhängigen Checkpointings ist der Job-Unit Checkpointer für die Verarbeitung der Checkpoint-Anfragen, Aktualisierung

der für die Wiederherstellung notwendigen Job Meta-Informationen und für das Ansprechen der ausgewählten Checkpointer zuständig. Darüber hinaus kümmert sich der Job-Checkpointer um die Erkennung der ausgefallenen Komponenten, um die Berechnung eines global konsistenten Zustandes und um die Wiederherstellung einer verteilten Anwendung (siehe Kapitel 3).
Folgender Abschnitt erläutert zunächst die Komponenten für die Verarbeitung der Checkpoint-Anfragen und für die Erstellung der Sicherungen. Anschließend folgt die Vorstellung der Abläufe für die Wiederherstellung der Anwendungen. Zunächst wird die Integration des Mechanismus zur Ausfallerkennung ausführlich erläutert. Die nachfolgenden zwei Abschnitte behandeln die Implementierung des in Kapitel 3 eingeführten Rollback-Propagation Algorithmus und die Wiederherstellung der ausgefallenen Komponenten.

4.4.1 Verarbeitung der Checkpoint-Anfragen

Der Job-Unit Checkpointer (CRExecMng) wurde im Rahmen dieser Arbeit erweitert. Somit ist es möglich die Checkpoint-Anfragen von der CRUC-Bibliothek entgegenzunehmen. Dafür wird während der Initialisierung des Job-Unit Checkpointers ein Server gestartet, der auf eingehende Client-Verbindungen wartet. Dieser Server befindet sich in einem eigenständigen Packet *jucapi* des Job-Unit Checkpointers und ist dafür zuständig, Anfragen der Clients entgegenzunehmen und zu bearbeiten. Der Server wird durch die Klasse *JUCApi_Server* initialisiert und erstellt für jede eingehende Clientanfrage einen eigenständigen Thread *JUCApi_CThread*. Jeder dieser Threads kümmert sich um die Kommunikation mit dem verbundenen Client. Dabei wird zunächst die Nachricht des Clients ausgelesen und verarbeitet.
Für das Zerlegen der empfangenen Nachricht ist der dafür entwickelte Parser *JUCApi_Parser* verantwortlich. Dieser Parser verarbeitet die Anfragen und speichert die empfangenen Werte zwischen. Anschließend wird ein Handler *JUCApi_Handler* für die Ausführung der gespeicherten Anfragen, aufgerufen. Derzeit stellt der Handler zwei in der Auflistung 4.17 dargestellte Methoden zur Verfügung.

Listing 4.17: Methoden des Handlers

```
1 public void handleMessage(Socket clientSocket, JUCApi_Parser message);
2 public void checkpoint_request(JUCApi_CPReqquest cp);
```

Die Methode *handleMessage()* ist für die Ausführung der empfangenen Anweisungen zuständig. Sie führt entsprechend der geparsten Anweisung eine Aktion aus. Im Falle des unabhängigen Checkpointings, erkennt der Handler ob es sich um eine Checkpoint-Anfrage handelt. Daraufhin wird ein Checkpoint-Request *JUCApi_CPReqquest* erstellt und die dafür vorgesehene *checkpoint_request()*-Routine aufgerufen. Diese Routine führt die für eine Sicherung der Anwendung notwendigen Routinen des Job-Unit Checkpointers aus. Anschließend wird die CRUC-Bibliothek über den Ausgang der Sicherung informiert. Der nachfolgende Abschnitt 4.4.2 erläutert diesen Vorgang etwas genauer.

4.4.2 Erstellung einer Sicherung

Der bestehende Execution-Manager wurde um eine weitere Methode *applicationInitiatedCheckpoint()* für die Erstellung der unabhängigen Sicherung erweitert. Im XtreemOS-Kontext wird der ExecMng für die Verwaltung der Job-Units verwendet. Dadurch enthält er unter anderen die Informationen über die Abbildung zwischen dem Job und dessen Job-Units (siehe Abschnitt 2.3.1).
Die im vorherigen Abschnitt 4.4.1 vorgestellten Komponenten zum Verarbeiten der Checkpoint-Anfragen rufen die oben genannte Methode auf, um eine unabhängige Sicherung auszulösen. Somit werden die vom ExecMng zur Verfügung gestellten Informationen über den Job (z.B. Job-ID) und dessen Prozessliste (PIDs) an die Routine *appInitiatedJobUnitChekcpoint()* des Job-Unit Checkpointers übergeben. Diese Routine wird für die Auswahl des richtigen Checkpointers und für die Erstellung einer Sicherung der Job-Unit verwendet. Der Sicherungsvorgang erfolgt nach der Auswahl des Checkpoints in drei Schritten.
Zunächst wird die Job-Unit angehalten. Anschließend erfolgt die Erstellung eines Checkpoints und die Aktualisierung der Job/Job-Unit Meta-Informationen auf einem persistenten Speicher. Zuletzt wird die angehaltene Job-Unit wieder gestartet und das Ergebnis der Sicherung der CRUC-Bibliothek mitgeteilt.
Für die Aktualisierung der Job/Job-Unit Meta-Informationen werden zwei weitere, neue Methoden im Job-Unit Checkpointer eingesetzt. Die erste Methode *updateJobMetaData()* ist für die Aktualisierung der Job-Meta Informationen aus der Datei *JobID.txt*, zuständig. Sie beinhaltet alle Job-Unit IDs des Jobs mit der jeweils aktuellsten Checkpoint-Version der Job-Unit und wird für die Identifizierung aller Job-Units eines Jobs im Rahmen des Wiederherstellungsvorgangs benötigt. Für die Aktualisierung der Job-Unit Meta-

Informationen ist die Routine *updateJobUnitMetaData()* zuständig. Ihre Aufgabe besteht in der Erstellung der Ordner für jeden neuen Checkpoint und in der Speicherung der Meta-Informationen über die Job-Units in Form einer XML-Datei. Diese Datei beinhaltet neben den Angaben über den Resourcenverbrauch, Informationen über die Checkpoint-Strategie, Checkpointer, Aufenthaltsort und einiges mehr. All diese Angaben werden für die Wiederherstellung der Job-Units verwendet (dazu mehr in Abschnitt 4.4.5).

4.4.3 Erkennung von Anwendungsausfällen

Folgender Abschnitt beschäftigt sich mit der Erkennung von Job-Unit Ausfällen. Wie bereits in Abschnitt 2.3.1 näher erläutert, ist der ExecMng für die Verwaltung und Überwachung der Job-Units eines Jobs zuständig. Dadurch ist er derzeit in der Lage einen Job-Unit-Ausfall zu erkennen und den Job-Manager über den Ausfall zu informieren. Der Job-Manager ist daraufhin in der Lage die von der Job-Unit reservierten Ressourcen freigeben.
Im Rahmen dieser Arbeit wäre jedoch eine weitere, derzeit noch nicht existierende Funktionalität zur Überwachung der ausgefallenen Job-Units von großer Bedeutung. Für eine erfolgreiche Wiederherstellung der ausgefallenen Job-Units, müssen alle Ausfälle im Job-Checkpointer (CRJobMng) festgehalten werden. Dafür wurde der Job-Checkpointer um einen Ausfalls-Monitor erweitert. Diese Komponente beinhaltet eine Hashtabelle *failureMonitor*, die zur Speicherung der Ausgefallenen Job-Units eines Jobs verwendet wird. In diesem Zusammenhang ordnet sie jeder Job-ID einen *MonitorEntry()*-Eintrag zu. Dieser Eintrag besteht aus einer weiteren Hashtabelle, die für jede JobUnit-ID eines Jobs den zuständigen Exit-Code bereithält. Die Abbildung 4.3 stellt den Aufbau des Ausfall-Monitors grafisch dar.

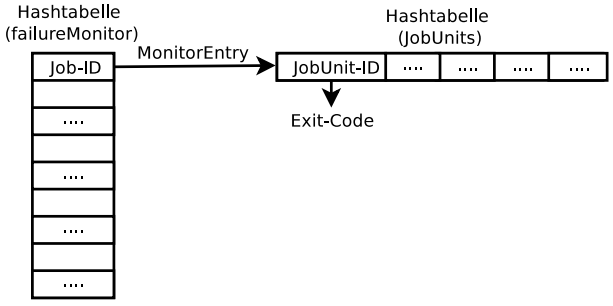

Abbildung 4.3: Aufbau des Ausfall-Monitors

Zum Einfügen und Löschen von Einträgen, stellt der Ausfalls-Monitor die in der Auflistung 4.18 dargestellten Methoden zur Verfügung. Beide Routinen werden sowohl vom ExecMng als auch vom Job-Checkpointer zur Behandlung der Job-Unit Ausfälle verwendet.

Listing 4.18: Überwachung der Job-Unit Ausfälle

```
1 public void reportFailedJobUnit(String jobId, Integer jobUnitId, Integer
      exitValue);
2 public void removeFailedJobUnit(String jobId, Integer jobUnitId);
```

Die Routine *reportFailedJobUnit()* wird beim unabhängigen Checkpointing vom ExecMng immer dann aufgerufen, wenn ein Job-Unit-Ausfall erkannt wird. Diese Routine ist für die Protokollierung des Ausfalls im Ausfalls-Monitor des Job-Checkpointers zuständig. Nach der erfolgreichen Wiederherstellung der ausgefallenen Job-Units, müssen die Einträge aus dem Monitor entfernt werden. Für diesen Vorgang ist die Routine *removeFailedJobUnit()* zuständig. Sie wird unmittelbar nach der Wiederherstellung der Job-Unit vom Job Checkpointer aus der Routine *resumeJobCompleteRST()* aufgerufen.

4.4.4 Berechnung eines global konsistenten Zustandes

Für eine erfolgreiche Wiederherstellung der ausgefallenen Job-Units ist die Berechnung eines global konsistenten Zustands der Anwendung notwendig. Dieser Vorgang wird mit Hilfe des in Kapitel 3 näher erläuterten Rollback-Propagation Algorithmus durchgeführt. Dazu wurde der Job-Checkpointer um eine weitere Komponente, genannt Recovery-Manager (RecMng) , erweitert. Diese Komponente ist für die Berechnung einer konsistenten Recovery-Line anhand der von CRUC-Bibliothek aufgezeichneten Abhängigkeiten zuständig. Der RecMng besteht aus zwei Hauptkomponenten.
Die erste Komponente *DependencyParser* ist für das Auslesen der Abhängigkeitsinformationen aus der XML-Datei *dep_list_vX.xml* bzw. *dep_list_current.xml* zuständig. Die Abhängigkeitsinformationen jeder Job-Unit werden ausgelesen und in einem Array mit den Einträgen vom Typ *DependencyEntry* gespeichert.
Die zweite Komponente ist der *RecLineCalculator*. Diese Komponente ist für die Berechnung der Recovery-Line zuständig. Sie implementiert den Rollback-Propagation Algorithmus und wird im Rahmen des Wiederherstellungsvorgangs für die Berechnung

des global konsistenten Zustandes der Anwendung verwendet. Die Auflistung 4.19 stellt die dafür nötigen, öffentlichen Aufrufe dar.

Listing 4.19: Berechnung der Recovery-Line

```
1 public void RecLineCalculator(Hashtable<Integer, DependencyEntry> rootSet);
2 public Integer triggerRecoveryLineCalculation();
```

Der Konstruktor von *RecLineCalculator()* erwartet als Parameter eine Hashtabelle der Abhängigkeitsinformationen der noch laufenden und bereits ausgefallenen Job-Units und hält diese in Form einer Member-Variablen fest. An dieser Stelle ist es wichtig zu erwähnen, dass die Hashtabelle im Initialzustand nur die zuletzt aufgezeichneten Abhängigkeitsinformationen der beteiligten Job-Units beinhalten muss. Sollten Aufgrund der ermittelten Abhängigkeiten, weitere Abhängigkeitsinformationen notwendig sein, so werden diese dynamisch nachgeladen.
Mit Hilfe der *triggerRecoveryLineCalculation()*-Routine kann die Berechnung der Recovery Line gestartet werden. Das Ergebnis dieser Berechnung ist eine modifizierte Hashtabelle, die aus den Einträgen besteht, die einen konsistenten Zustand der Anwendung darstellen.
Zum Ansprechen der Recovery-Line Komponente wurde der Job-Checkpointer um eine weitere Methode *calculateRecoveryLine()* erweitert. Diese Methode erwartet als Parameter eine Hashtabelle mit den Abhängigkeitsinformationen. Sie ruft anschließend die entsprechen Routinen des Moduls zur Berechnung der Recovery-Line auf.

4.4.5 Wiederherstellung der ausgefallenen Komponenten

Dieser Abschnitt behandelt die Wiederherstellung der ausgefallenen Komponenten. Dafür müssen als erstes die in Abschnitt 4.4.2 aufzeichneten Job Meta-Informationen ausgelesen werden. Anschließend erfolgt die in Abschnitt 4.4.3 vorgestellte Erkennung der ausgefallenen Job-Units. Schließlich wird das in Abschnitt 4.4.4 vorgestellte Modul zur Berechnung der Recovery-Line aufgerufen. Nach einer erfolgreichen Berechnung der Recovery-Line, findet die Wiederherstellung der ausgefallenen Job-Units statt.
Oben genannte Bestandteile wurden im Rahmen dieser Arbeit in die XtreemGCP Komponente unter der Bewahrung ihrer Architektur integriert. Die erste Änderung betrifft die vom AEM-Framework zur Verfügung gestellten *XConsole*. Mit Hilfe dieser Konsole können z.B. Anwendungen im Grid ausgeführt werden. Somit ist es im XtreemGCP-Kontext

möglich, mittels des Befehls *xcheckpoint* eine koordinierte Sicherung vorzunehmen. Aufgrund der Tatsache, dass im Rahmen des unabhängigen Checkpointings die Job-Units selbst in der Lage sind, Sicherungen vorzunehmen, ist der oben genannte Befehl für diese Abhandlung weniger relevant. Die Wiederherstellung der Anwendung erfolgt im Fehlerfall benutzerinitiiert. Dazu wird der Befehl *xrestart* verwendet. Job-ID und Version müssen dabei angegeben werden.
Der Restart-Befehl wurde um die Funktionalität zum unabhängigen Checkpointing erweitert. Bei fehlender Versionsangabe geht der XtreemGCP von einer unabhängig gesicherten Anwendung aus. Anschließend wird die Wiederherstellung mit Hilfe der Job-Checkpointer Routine *restartJobInitUC()* eingeleitet. Dabei wird zunächst die *MultiJobControl (MJC)*-Struktur des Grid-Checkpointers initialisiert. Diese Struktur enthält unter anderem die ausgewählte Checkpoint-Strategie (z.B. unabhängiges Checkpointing). Anschließend wird die in Abschnitt 4.4.2 angelegte Datei *JobID.txt* mit den Job Meta-Informationen ausgelesen. Der Job-Checkpointer erhält dadurch die Job-Unit IDs und die aktuellsten Checkpoint-Versionen der jeweiligen Job-Units. Diese Informationen werden zum Auslesen der Job-Unit Meta-Informationen und zum Eintragen der Ressource-Adressen auf denen die Job-Units zuvor liefen, in die *MJC*-Struktur, benutzt. Dadurch werden dem Job-Checkpointer die Grid-Knoten bekannt, auf denen die Wiederherstellung der Job-Units erfolgen soll. Es folgt schließlich ein Aufruf der nativen *processWithRestart()*-Routine zum Fortsetzen der Wiederherstellung.
Diese Routine stellt sicher, dass jede Job-Unit einem Grid-Knoten zugeordnet wird und ruft anschließend eine Methode *readRestartMetaData()* zum Auslesen der Job-Unit Meta-Informationen auf. An dieser Stelle wird zunächst die Checkpoint-Strategie überprüft. Im Falle des unabhängigen Checkpointings, wird die Routine *initIndependentRST()* zum Erkennen der ausgefallenen Job-Units und zum Berechnen der Recovery-Line ausgeführt. Die *MultiJobUnitControl*-Struktur des XtreemGCP wurde um eine weitere Hashtabelle *recLine* zum Bereithalten der Recovery-Line erweitert. Nach der Initialisierung dieser Hashtabelle erfolgt die Berechnung der Recovery-Line unter Berücksichtigung des in Abschnitt 4.4.3 dargestellten Ausfall-Monitors.
Sollten keine Ausfallinformationen zum Job vorhanden sein, so erfolgt eine Berechnung der Recovery-Line ab dem letzten Checkpoint aller Job-Units. Dies ist immer dann der Fall, wenn alle Grid-Knoten ausfallen sollten. Liegt nur ein Teilausfall der Grid-Knoten vor, so werden die aktuellen Abhängigkeiten der noch laufenden Job-Units und die des letzten Checkpoints der ausgefallenen Job-Units ausgelesen und in der Recovery-Line Berechnung mitberücksichtigt.

Für eine erfolgreiche Wiederherstellung der Anwendung müssen in der aktuellen Implementierung zunächst die Server und anschließend die Clients wiederhergestellt werden (siehe Abschnitt 3.5). Dafür werden nach der Recovery-Line Berechnung zwei weitere Hashtabellen *clientHash* und *serverHash* mit Hilfe der *generateServerClientTables()*-Routine initialisiert. Beide Hashtabellen werden mit den mittels der Recovery-Line-Berechnung ermittelten Informationen gefüllt. Diese Hashtabellen dienen der Trennung von Client und Server Job-Units während der Wiederherstellungsphase.
Nach der Berechnung der Recovery-Line und nach der Initialisierung der Client/Server-Hashtabellen, kann der Wiederherstellungsvorgang gestartet werden. Beim unabhängigen Checkpointing müssen die in der Tabelle 4.2 dargestellten Fälle beachtet werden.

Szenarien	**Ausfalls-Monitor**	**Aktion**
RecVersion $<$ LastVersion	Nicht vorhanden	Restart von RecVersion
RecVersion $<=$ LastVersion	Vorhanden	Kill Job-Unit und Restart von RecVersion
RecVersion $>$ LastVersion	Vorhanden	Überspringe Job-Unit und Aktualisiere die Client-/Server Hashtabelle
RecVersion $=$ LastVersion	Nicht Vorhanden	Restart von RecVersion

Tabelle 4.2: Unabhängiges Checkpointing, Wiederherstellung

Das Feld Szenario stellt den jeweiligen Fall dar. Mit der *RecVersion* Angabe wird die von der Recovery-Line vorgeschlagene Version für die Wiederherstellung bezeichnet. Die *LastVersion* gibt die letzte Version des Checkpoints einer Job-Unit an. Der Ausfall-Monitor gibt an, ob die Job-Unit derzeit ausgeführt wird. (Vorhanden oder Nicht Vorhanden). Im Feld Aktion befindet sich die für das dargestellte Szenario zuständige Aktion.
Zunächst wird der Job mit Hilfe der XtreemGCP-Routinen auf der Grid-Ebene wiederhergestellt. Anschließend folgt die Wiederherstellung der Job-Units unter Beachtung der dargestellten Szenarien. Dabei werden zunächst alle Job-Units mittels der bestehenden Routinen des XtreemGCP wiederaufgebaut. Da in der aktuellen Implementierung zunächst die Server und anschließend die Clients gestartet werden müssen, wurde die XtreemGCP-Sequenz an dieser Stelle um eine zusätzliche Methode *resumeJobRST_UC()* erweitert. Sie wird immer dann aufgerufen, wenn die Wiederherstellung der Job-Units im Kontext des unabhängigen Checkpointings stattfinden sollte. Ihre Aufgabe besteht darin zunächst die Server und anschließend die Clients mit Hilfe der vorher aufgebauten Client/Server-Hashtabellen zu starten.

4.5 Adaptives Umschalten der Checkpoint-Strategien

In diesem Abschnitt wird der erste Prototyp der *LibSwitch*-Bibliothek vorgestellt. Diese Bibliothek wird während der Installation der CRUC-Bibliothek eingerichtet und soll in der Zukunft den dynamischen Wechsel der Checkpoint-Strategien ermöglichen.
Es wird grundsätzlich zwischen zwei Checkpoint-Strategien unterschieden: Koordiniert und Unkoordiniert. Für das koordinierte Checkpointing existiert bereits eine an der Heinrich-Heine-Universität (HHU) entwickelte Bibliothek, genannt CSCC (Consistent Snapshots of Communication Channels) [Sli09]. Mit ihrer Hilfe können die verteilten Anwendungen, die mittels Nachrichtenaustauschs miteinander kommunizieren, konsistent gesichert und wiederhergestellt werden. Im Rahmen dieser Arbeit wurde eine weitere Bibliothek namens CRUC (Checkpoint/Restart Uncoordinated) entwickelt, die den Anwendungen erlaubt, Sicherungen unabhängig voneinander zu erstellen.
Sowohl von der CSCC- als auch von der CRUC-Bibliothek müssen bestimmte Systemaufrufe (z.B. send, recv, socket, bind, etc.) abgefangen und manipuliert werden. Für einen dynamischen Wechsel der Checkpoint-Strategie bedarf es somit einer weiteren Bibliothek, die für das Abfangen und Weiterleiten dieser Systemaufrufe an die derzeit ausgewählte Checkpoint-Bibliothek zuständig ist.
Zu diesem Zweck wurde die in der Abbildung 4.4 dargestellte *LibSwitch*-Bibliothek entwickelt. Ihre Aufgabe besteht darin die Systemaufrufe abzufangen, eine Bibliothek (z.B. LibCRUC) dynamisch zu laden und die Systemaufrufe an die geladene Bibliothek weiterzuleiten.

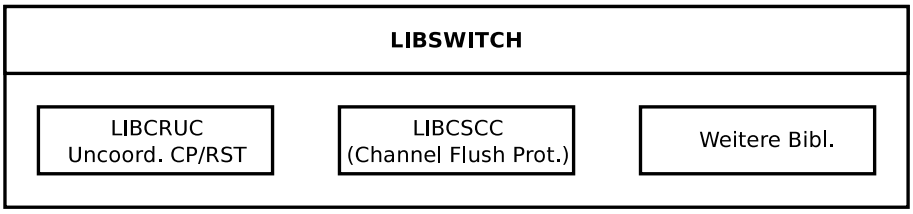

Abbildung 4.4: Umschalten der Checkpoint-Strategien, LibSwitch Bibliothek

Die *LibSwitch* wird mittels des in Abschnitt 4.1 vorgestellten LD_PRELOAD-Mechanismus mit jeder Anwendung automatisch geladen und ist für das Abfangen der festgelegten Systemaufrufe zuständig. Die derzeitige Implementierung der *LibSwitch* wählt als Ausgangsstrategie das unabhängige Checkpointing aus und leitet alle *send, recv und bind*-Aufrufe an die entsprechenden Routinen der *LibCRUC* weiter. Zum Abfangen weiterer Systemaufrufe anderer Bibliotheken, kann die *LibSwitch* auf eine einfache Art

und Weise ergänzt werden.
Zum Zwecke eines dynamischen Wechsels der Checkpoint-Strategie, sieht die Bibliothek derzeit eine Benachrichtigung über ein Signal/Message-Queue Kombination vor. Dabei wird von dem Aufrufer (XtreemGCP) zuerst eine Message-Queue erstellt und eine Nachricht mit der gewünschten Checkpoint-Strategie abgeschickt. Anschließend benachrichtigt er die *LibSwitch* mittels eines freien Echtzeitsignals über einen anstehenden Strategie-Wechsel. In dem entsprechen Signal-Handler liest die Bibliothek schließlich aus der erstellten Message-Queue die Nachricht aus und führt einen Wechsel der Strategie durch. Da die *LibSwitch* im Adressraum eines jeden Prozesses läuft, muss sie während des Checkpointvorgangs mitgesichert werden. Dies ist aufgrund des Signal-Handler Ansatzes möglich, da die Sicherung der Signal-Handler von den meisten derzeit verfügbaren Checkpointer (z.B. BLCR, LinuxSSI, OpenVZ, DMTCP, Libckpt, Condor, etc.) - Implementierungen unterstützt werden.
Abschließend ist es wichtig zu erwähnen, dass die Checkpoint-Bibliotheken (z.B. CRUC, CSCC, etc.) unterschiedliche Anforderungen an einen bevorstehenden Strategie-Wechsel stellen. Dadurch muss z.B. bei CRUC vor einem anstehenden Strategie-Wechsel sichergestellt werden, dass die CRUC-Bibliothek vor dem Nachrichtenaustausch bei allen beteiligen Prozessen zuerst geladen wird. Falls dies nur bei dem Absender-Prozess geschieht, werden zwar Nachrichten manipuliert und gesendet, können jedoch auf der Empfänger-Seite nicht richtig interpretiert werden. Analog verhält es sich, falls die CRUC-Bibliothek nur beim Empfänger geladen wurde. Dadurch verschickt der Sender-Prozess unmodifizierte Nachrichten, die der Empfänger nicht interpretieren kann. In der derzeitigen Implementierung existiert noch kein Mechanismus, welches einen solchen Fall behandelt. Dadurch wird die Checkpoint-Strategie derzeit noch statisch festgelegt.

Kapitel 5

Leistungsbewertung und Analyse

Folgendes Kapitel beschäftigt sich mit der Performance-Evaluierung der in Kapitel 4 vorgestellten Implementierung. In diesem Zusammenhang wird in Abschnitt 5.1 die heterogene Testumgebung, bestehend aus einem SSI-Cluster und vier gewöhnlichen PC-Knoten erläutert. Anschließend wird die im Rahmen dieser Arbeit für die Überprüfung der Implementierung entwickelte Testanwendung vorgestellt. Der nachfolgende Abschnitt 5.2 widmet sich der Performance-Evaluierung. In diesem Abschnitt wird die Implementierung mittels der Testanwendung verifiziert und auf ihre Performance hin evaluiert. Zuletzt folgt in Abschnitt 5.3 eine Zusammenfassung der Ergebnisse.

5.1 Ausgangssituation

5.1.1 Testumgebung

In diesem Abschnitt wird zunächst die Testumgebung und anschließend die im Rahmen dieser Arbeit entwickelte Testanwendung vorgestellt. Die Testumgebung besteht aus insgesamt sechs *AMD Opteron 244*-Knoten, die mit jeweils *1800MHz* getaktet sind. Darüber hinaus kommt auf jedem der Knoten das Betriebssystem Debian GNU/Linux in der aktuellsten, stabilen Version 5.0 („Lenny“) zum Einsatz. Alle Knoten verfügen außerdem über *1024MB*-Arbeitsspeicher, *1024MB*-Auslagerungsspeicher und eine Gigabit Netzwerkkarte. Die ersten vier Knoten sind gewöhnliche PC-Knoten. Auf diesen Knoten

wird der Vanilla-Kernel in der Version *2.6.20.20* und eine im Rahmen von XtreemOS modifizierte BLCR *0.8.0*-Variante eingesetzt. Die restlichen zwei Knoten benutzten den aktuellsten LinuxSSI 1.0-rc2 Kernel und formen damit einen SSI-Cluster. Die oben genannte Konfiguration der Testumgebung kann den Tabellen 5.1 und 5.2 entnommen werden.

Prozessor:	AMD Opteron 244, 1800 MHz
Arbeitsspeicher:	2048 MB
Auslagerungsspeicher:	2048 MB
Netzwerkkarte:	Gigabit
Betriebssystem:	Debian GNU/Linux 5.0 („Lenny“)
Kernel:	LinuxSSI 1.0-rc2
Checkpointer:	Integriert in LinuxSSI

Tabelle 5.1: Konfiguration der LinuxSSI-Knoten

Prozessor:	AMD Opteron 244, 1800 MHz
Arbeitsspeicher:	2048 MB
Auslagerungsspeicher:	2048 MB
Netzwerkkarte:	Gigabit
Betriebssystem:	Debian GNU/Linux 5.0 („Lenny“)
Kernel:	2.6.20.20 (Vanilla)
Checkpointer:	BLCR, Version 0.8.0 (XtreemOS)

Tabelle 5.2: Konfiguration der BLCR-Knoten

Alle sechs Knoten benutzen ein mittels des *Network File System (NFS)* eingebundenes Root-Dateisystem. Dieses Dateisystem wird von einem zusätzlichen, siebten Knoten, dem sogenannten Master-Knoten zu Verfügung gestellt. Die Aufgabe des Master-Knotens besteht darin den Slave-Knoten (LinuxSSI, BLCR) einen netzwerkbasierten Bootvorgang zu ermöglichen. Mit Hilfe dieses Vorgangs können die Slave-Knoten ohne Festplattenspeicher über ein Netzwerk das Betriebssystem laden. Das soeben erläuterte Verfahren wird als *Preboot eXecution Environment (PXE)* bezeichnet und kommt in dieser Arbeit zum Einsatz.
Nach einem erfolgreichen Bootvorgang wird auf allen Knoten eine im Rahmen dieser Arbeit für unabhängiges Checkpointing erweiterte Version der AEM geladen. Diese AEM-Version erlaubt es den Anwendungen unabhängige Sicherungen vorzunehmen. AEM nimmt auf der Grid-Ebene die Anfragen entgegen, wählt den richtigen Checkpointer (LinuxSSI oder BLCR) aus und nimmt die Sicherungen vor. Die nachfolgende Abbildung 5.1 stellt den Aufbau der Testumgebung grafisch dar.

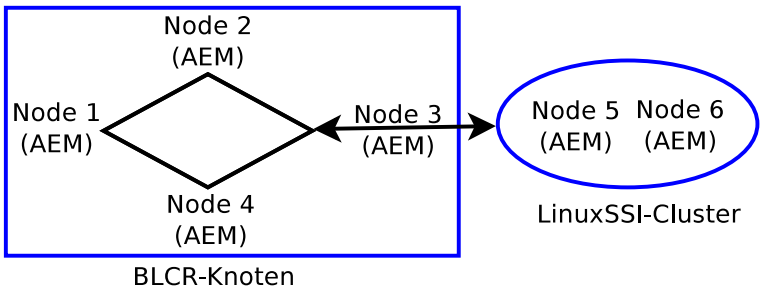

Abbildung 5.1: Aufbau der Testumgebung

Wie der Abbildung 5.1 zu entnehmen, wird der LinuxSSI-Cluster hierbei durch zwei AEM-Instanzen repräsentiert. Im Kontext von XtreemOS kommt jedoch nur eine AEM-Instanz pro Cluster zum Einsatz. Die zweite AEM-Instanz wurde im Rahmen dieser Testumgebung lediglich zur optimalen Ausnutzung der verfügbaren Knoten hinzugenommen, da sonst ein Knoten ungenutzt geblieben wäre.

5.1.2 Testanwendung

In diesem Abschnitt wird die Arbeitsweise der entwickelten Testanwendung erläutert. Für die Überprüfung der Korrektheit der Implementierung wurde eine Client-/Server Anwendung entwickelt. Diese Anwendung wird zunächst statisch gegen die in Abschnitt 4 vorgestellte CRUC-API Bibliothek gelinkt. Dadurch ist die Anwendung in der Lage, unabhängige Sicherungen vorzunehmen.
Zusätzlich müssen während der gesamten Laufzeit der Anwendung, die Abhängigkeiten verwaltet und aufgezeichnet werden. Dazu wird eine weitere, ebenfalls in Abschnitt 4 vorgestellte Bibliothek dynamisch in den Adressraum der Anwendung geladen.
Für die Ausführung der Anwendung innerhalb der Testumgebung wird zunächst der Server mit Hilfe der *xconsole* von XtreemOS gestartet. Eine initiale Sicherung wird erstellt. Anschließend folgt die Ausführung der Clients. Dabei stellt jeder Client eine TCP/IP-Verbindung zum Server her und führt ebenfalls eine initiale Sicherung durch. Nach der Durchführung der Sicherungsvorgänge von dem Server und von den Clients, wird zunächst eine Zahl $X \geq 0$ im Bereich $[0, Y-1]$ mit $Y \geq 0$ gewählt. Anschließend wählt jeder Client eine Zufallszahl zwischen $[0, Y-1]$ und vergleicht diese Zahl mit dem Wert von X. Im Falle der Übereinstimmung beider Werte, wird eine Sicherung vorgenommen, ansonsten wird lediglich eine positive Zahl verschickt. Nach jedem Sende-

Vorgang erfolgt die Inkrementierung dieser Zahl. Nach dem Empfang der von dem Client gesendeten Zahl, wählt der Server ebenfalls eine Zufallszahl. Anhand dieser Zahl wird entschieden, ob eine Sicherung vorgenommen werden muss. Im Rahmen dieser Arbeit wurde der Wert für *X* auf *10* festgelegt.
Zusätzlich zu der Zufälligkeit einer Sicherung müssen die Ausfälle simuliert werden. Dieser Vorgang kann entweder explizit von außerhalb der Anwendung mit Hilfe des *kill*-Befehls oder implizit von der Anwendung selbst mittels eines *exit*-Befehls gestartet werden. Für die implizite Absturzsimulation, wurde eine ähnliche Vorgehensweise wie für die zufällige Erstellung einer Sicherung verwendet. Im Gegensatz dazu wird keine Sicherung vorgenommen, sondern eine Terminierung des Programms mit dem *exit*-Befehl eingeleitet.
Die Abbildung 5.2 stellt ein mögliches Kommunikationsmuster der soeben beschriebenen Testanwendung mit *n*-Clients grafisch dar.

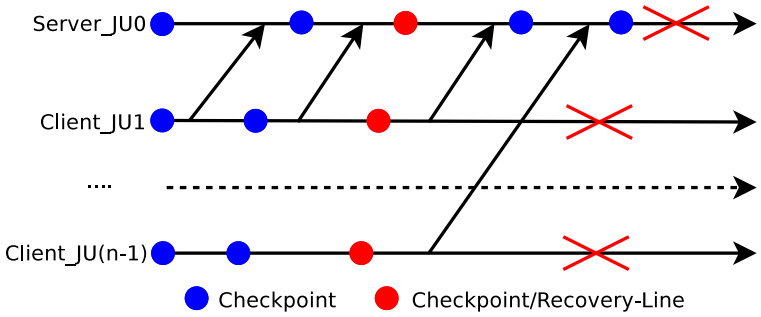

Abbildung 5.2: Kommunikationsmuster der Testanwendung

Der nachfolgende Abschnitt 5.2 erläutert die Messergebnisse, die mit Hilfe der oben genannten Testanwendung ermittelt wurden. Dabei beziehen sich die durchgeführten Messungen auf die Dauer der Aufzeichnung von Abhängigkeiten, auf die Sicherungs- und Wiederherstellungszeit, so wie auf die Zeit zur Berechnung einer Recovery-Line.

5.2 Messungen

In diesem Abschnitt werden die ermittelten Messergebnisse präsentiert. Die Messungen wurden mit Hilfe der in Abschnitt 5.1 vorgestellten Client-/Server Testanwendung, innerhalb der aufgebauten Testumgebung mit sechs Job-Units durchgeführt. Anschließend wurden die Zeiten für die Verwaltung der Abhängigkeiten, Erstellung einer unabhängigen

Sicherung, Berechnung der Recovery-Line, so wie die Wiederherstellung der Testanwendung ermittelt. Die zu diesem Zweck verwendete Messreihe bestand aus zehn Messungen. Für die Berechnung eines Durchschnittswerts, wurde schließlich ein arithmetisches Mittel jeder Messreihe gebildet. Die ermittelten Messdaten werden in den nachfolgenden Absätzen 5.2.2 und 5.2.3 ausführlich erläutert.

5.2.1 Verwaltung der Abhängigkeitsinformationen

Im Rahmen dieser Arbeit wurden die Aufrufe der nativen Sende- und Empfangsroutinen abgefangen und modifiziert. Dadurch konnten ausgehende Nachrichten manipuliert und mit den Abhängigkeitsinformationen versehen werden. Diese Informationen wurden bei der anschließenden Berechnung der Recovery-Line dazu verwendet, um einen global konsistenten Zustand der Anwendung zu ermitteln. Im Laufe der Messungen zeigte sich, dass der durch die Modifizierung der Sende- und Empfangsroutinen entstandene Aufwand sich nur marginal von der Verwendung der nativen Aufrufe unterscheidet und unter einer Mikrosekunde liegt. Dies hängt damit zusammen, dass keine Nachrichten zum Anhängen der Abhängigkeitsinformation umkopiert werden müssen. Nach dem Abfangen der Sendeaufrufe, werden zunächst die Abhängigkeitsinformationen und anschließend die eigentlichen Nachrichten übertragen (siehe Kapitel 4).

5.2.2 Erstellung einer unabhängigen Sicherung

Die Abbildung 5.3 stellt den ermittelten Durchschnittswert einer unabhängigen Sicherung für BLCR und LinuxSSI grafisch dar. Es zeigt sich, dass eine unabhängige Sicherung mit BLCR um den Faktor 4 langsamer ist als mit LinuxSSI. Dies hängt mit der internen Funktionsweise der BLCR Translation-Library zusammen. Während die Routinen des LinuxSSI-Checkpointers direkt, mittels der *ioctl*-Aufrufe angesteuert werden können, findet bei BLCR eine Synchronisierung des Checkpoint-Vorganges mit Hilfe von Message-Queues, Semaphoren und lokalen Dateien statt. Dadurch wird die Erstellung einer Sicherung verlangsamt.

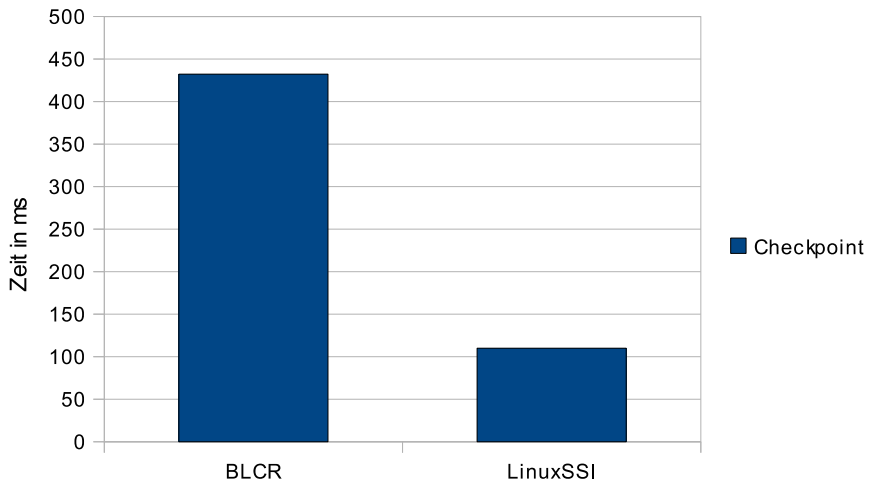

Abbildung 5.3: Durchschnittliche Dauer für die Erstellung einer Sicherung

5.2.3 Wiederherstellung der Anwendung

Für die Messung der Wiederherstellungszeit der Anwendung wurden zwei Messzenarien definiert. Das erste Messzenario betrachtet lediglich die Berechnungszeit der Recovery-Line. Bei dem zweiten Szenario hingegen wird sowohl die Berechnungszeit der Recovery-Line, als auch die Wiederherstellungszeit der Job-Units berücksichtigt. Die Abbildung 5.4 stellt die durchschnittliche Dauer für die Berechnung der Recovery-Line grafisch dar.

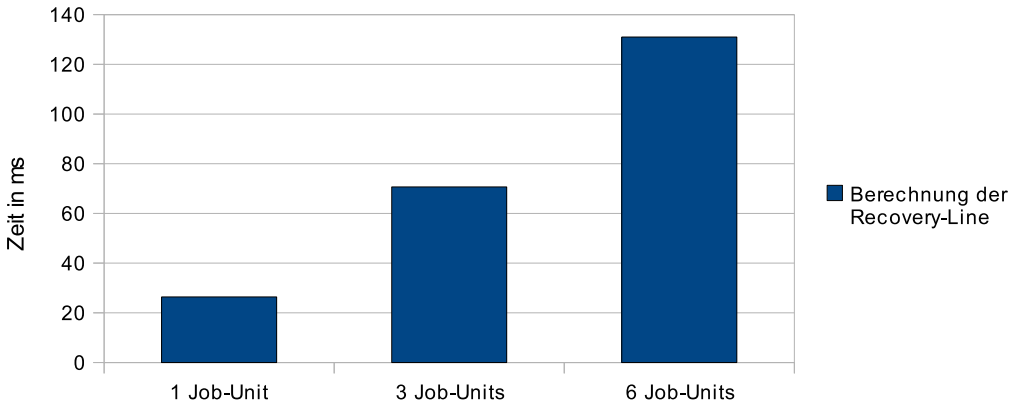

Abbildung 5.4: Durchschnittliche Dauer für die Berechnung der Recovery-Line

Die Berechnung der Recovery-Line für eine Job-Unit dauerte *26,4ms*. Dies hängt damit zusammen, dass für die Berechnung der Recovery-Line zunächst eine initiale Menge von Abhängigkeitsinformationen aufgebaut werden muss. Da bei einer

Job-Unit die Recovery-Line sofort terminiert, spiegelt die dargestellte Zeit lediglich das Zusammenstellen der Abhängigkeitsinformationen wieder. Beim Einsatz der drei Job-Units, beträgt die Zeit *70,67ms*. Dies ist darauf zurückzuführen, dass zunächst eine Menge von Abhängigkeitsinformationen für drei Job-Units aufgebaut werden muss. Anschließend erfolgt die Berechnung der Recovery-Line. Die Zeit für die Berechnung der Recovery-Line ist sowohl vom Kommunikationsmuster der Anwendung als auch von der Frage wie weit die Anwendung zurückgerollt werden muss, abhängig. Aufgrund des nicht deterministischen Verhaltens der Testanwendung (siehe Abschnitt 5.1.2) ist dieses Kommunikationsmuster nicht eindeutig. Deswegen stellen die dargestellten Messungen lediglich die Zeit für das zum Wiederherstellungszeitpunkt geltende Kommunikationsmuster dar. Analog verhält es sich mit der Wiederherstellung einer aus sechs Job-Units bestehenden Anwendung. Die hierbei ermittelte Zeit ist mit *131ms* nahezu doppelt so groß wie für die oben genannten drei Job-Units. Dies hängt damit zusammen, dass hierbei dreifach so viele Abhängigkeitsinformationen geladen werden müssen. Zusätzlich kommt noch die eigentliche Berechnung der Recovery-Line hinzu. Bei keiner der vorgestellten Messungen trat ein Kommunikationsmuster auf, welches einen Domino-Effekt nach sich ziehen, und dadurch die Anwendung in den initialen Zustand zurückversetzen könnte. Dies ist anhand der ermittelten Zeiten zu erkennen, die nahezu proportional zur der Anzahl der Job-Units ansteigen.

In der Abbildung 5.5 wird die durchschnittliche Dauer für die Wiederherstellung der Job-Units dargestellt.

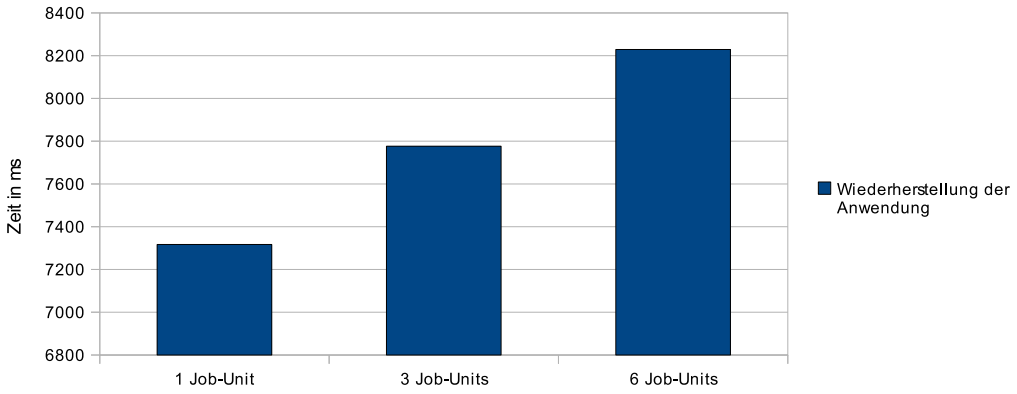

Abbildung 5.5: Durchschnittliche Dauer für Wiederherstellung der Anwendung

Wie in der Abbildung zu sehen, dauert die Wiederherstellung einer Job-Unit *7,3sec*. Diese Zeit beinhaltet sowohl die Berechnung der Recovery-Line, als auch die Wiederherstellung

der Job-Unit. Die Zeit für die Wiederherstellung von drei Job-Units, beträgt *7,7sec*. Dies hängt damit zusammen, dass die Wiederherstellung der Job-Units in XtreemGCP parallel stattfindet. Die durchschnittliche Dauer für die Wiederherstellung von sechs Job-Units beträgt *8,1sec*. Die ermittelten Zeiten besagen, dass die Zeit für die Wiederherstellung der Testanwendung linear mit der Anzahl der Job-Units ansteigt.

5.3 Schlussfolgerung

In diesem Abschnitt wurde die Performance der im Rahmen dieser Arbeit entwickelten Implementierung für unabhängiges Checkpointing evaluiert. Es lässt sich zusammenfassend sagen, dass die Performance der Implementierung stark von der Integration der Checkpointer abhängt. Dies machte sich insbesondere beim unabhängigen Sichern der Job-Units unter der Benutzung von BLCR bemerkbar. Die Zeiten waren hier um den Faktor 4 langsamer als unter LinuxSSI.
Der Aufwand für die Verwaltung der Abhängigkeiten lag sowohl beim Senden, als auch beim Empfangen der Nachrichten unter einer Mikrosekunde. Somit beeinflusst die Verwaltung der Abhängigkeitsinformation die Performance des Systems nur marginal.
Bei der Berechnung der Recovery-Line spielen zwei Faktoren eine wichtige Rolle. Zunächst muss eine initiale Menge aus Abhängigkeitsinformationen erzeugt werden. Die Dauer für die Erzeugung dieser Ausgangsmenge hängt von der Anzahl der Job-Units und deren Abhängigkeiten ab. Nach dem Aufbau der Ausgangsmenge, findet die Berechnung der Recovery-Line statt. Die Dauer dieser Berechnung hängt vom Kommunikationsmuster der Anwendung ab und verlangsamt sich im Falle eines Domino-Effekts. Es zeigte sich, dass für die im Rahmen dieser Arbeit entwickelte Testanwendung, die Zeit für die Berechnung der Recovery-Line proportional mit der Anzahl der Job-Units anstieg. Dies hängt mit der Tatsache zusammen, dass bei dem Kommunikationsmuster der Testanwendung kein Domino-Effekt auftrat.
Während bei der Berechnung der Recovery-Line die oben genannten Faktoren eine wichtige Rolle spielen, kommen bei der Wiederherstellung der Job-Units die XtreemGCP-Routinen zum Einsatz. Mittels der Messungen wurde ermittelt, dass die Zeit mit der steigenden Anzahl der Job-Units linear ansteigt.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Im Rahmen dieser Masterarbeit wurde ein Konzept zum unabhängigen Sichern einer verteilten Anwendung erarbeitet und erfolgreich in die bestehenden XtreemOS Grid-Checkpointer Komponente integriert. Dadurch kann entweder der Anwendungsentwickler oder der XtreemGCP-Dienst selbst entscheiden, zu welchem Zeitpunkt eine Sicherung der Anwendung vorgenommen werden soll. Bei den datenintesiven Anwendungen kann diese Funktionalität dazu benutzt werden, um den Koordinierungsaufwand zu vermindern, indem die Sicherungen ausschließlich an den Zeitpunkten vorgenommen werden, an denen die zu sichernde Datenmenge gering ist.
Aufgrund der möglichen Inter-Prozess Abhängigkeiten, die durch der Nachrichtenkommunikation einer verteilten Anwendung entstehen können, wurde eine dedizierte Bibliothek CRUC (Checkpoint/Restart Uncoordinated) entwickelt, die sich um die Aufzeichnung der Abhängigkeitsinformation kümmert. Ihre Aufgabe besteht darin die Abhängigkeiten zu empfangen und auf einem persistenten Speicher im XML-Format festzuhalten. Diese Bibliothek wird in den Adressraum eines jeden Prozesses der Anwendung dynamisch geladen und sorgt für die transparente Aufzeichnung der Abhängigkeiten.
Darüber hinaus wurde eine weitere dynamische Bibliothek, genannt CRUC-API entwickelt, die dem Anwendungsentwickler eine Sicherungsschnittstelle zur Verfügung stellt. Diese Bibliothek spricht die von der CRUC-Bibliothek exportierten Schnittstellen zur der Anwendungssicherung dynamisch an und sorgt schließlich dafür, dass der Grid-Checkpointer über eine anstehende Sicherung benachrichtigt wird.

Der Grid-Checkpointer verarbeitet die Anfragen der CRUC-Bibliothek und kümmert sich um die Durchführung der Anwendungssicherung. Dabei wird zunächst der Grid-Knoten spezifische Checkpointer (BLCR, LinuxSSI, OpenVZ, etc.) ausgewählt. Nach der Auswahl des Checkpointers findet schließlich die Sicherung des Prozesses und eine Aktualisierung der für eine Wiederherstellung der verteilten Anwendung notwendigen Meta-Informationen auf einem persistenten Speicher (z.B. XtreemFS) statt.
Eine verteilte Anwendung besteht in der Regel aus mehreren Prozessen, die ihrerseits auf unterschiedlichen Grid-Knoten ausgeführt werden. Diese Prozesse können entweder durch einen Knotenausfall oder aufgrund der internen Fehler oder der äußeren Einflüssen ausfallen. Für eine Wiederherstellung der ausgefallenen Prozesse müssen zunächst die Ausfälle erkannt werden. Dafür wurde der Grid-Checkpointer um einen Ausfallmonitor erweitert. Dieser Monitor erkennt und protokolliert die Ausfälle für eine spätere Wiederherstellung der Anwendung.
Während der Wiederherstellung muss ein global konsistenter Zustand der Anwendung berechnet werden. Dafür wurde der Rollback-Propagation Algorithmus [WF93] in den Grid-Checkpointer integriert. Dieser Algorithmus bedient sich der aufgezeichneten Abhängigkeitsinformationen so wie der Information aus dem Ausfallmonitor und ermittelt anhand dessen einen global konsistenten Zustand (Recovery-Line) der Anwendung. Dieser Zustand wird schließlich dafür Benutzt, um die Anwendung in einen konsistenten Zustand zu versetzen.
Die Implementierung wurde anschließend in einer heterogenen Umgebung bestehend aus sechs Knoten (BLCR und LinuxSSI), anhand einer im Rahmen dieser Arbeit entwickelten Client-/Server Anwendung auf ihre Performance hin evaluiert. Aufgrund der ermittelten Messergebnisse lässt sich zusammenfassend sagen, dass die unabhängige Sicherung unter BLCR um den Faktor 4 langsamer verlief als unter LinuxSSI. Dies hing mit der internen Funktionsweise der BLCR Translation-Library zusammen. Der Aufwand für die Verwaltung der Abhängigkeiten lag unter einer Mikrosekunde. Bei der Wiederherstellung zeigte sich, dass die Wiederherstellungsdauer der Testanwendung linear mit der Anzahl der Prozesse anstieg.
Die Messergebnisse belegten die Tauglichkeit der ausgearbeiteten Lösung zur unabhängigen Sicherung einer verteilten Anwendungen in einer heterogenen Grid-Umgebung und zeigten gleichzeitig weitere Optimierungsmöglichkeiten der Implementierung auf (siehe Abschnitt 6.2).

6.2 Ausblick

Im Verlauf dieser Arbeit stellten sich einige Optimierungs- und Erweiterungsmöglichkeiten heraus. Derzeit muss sowohl die ID als auch der JSDL-Pfad der Job-Unit aufgrund von Einschränkungen der AEM von der CRUC-Bibliothek aus übergeben werden. Dies kann durch die Einführung einer ID zur Verwaltung der Job-Units innerhalb der AEM verbessert werden. Darüber hinaus werden derzeit alle aufgezeichneten Abhängigkeitsinformationen jedes Prozesses in den dafür angelegten Ordnern im verteilten Speicher gespeichert. Die Ordnernamen werden aus den PIDs der jeweiligen Prozesse gebildet. Bei dieser Vorgehensweise kann es zu Konflikten mit den PIDs der auf unterschiedlichen Grid-Knoten laufenden Anwendungen kommen. In diesem Fall würden sich die Abhängigkeitsinformationen gegenseitig überschreiben. Die Lösung dieses Problems bedarf ebenfalls der Einführung einer Job-Unit ID. Allerdings ist diese ID nur auf der Grid-Ebene sichtbar und dadurch innerhalb der CRUC-Bibliothek nicht verfügbar. Dazu sollte die CRUC-Bibliothek während ihrer Initialisierung eine Job/Job-Unit ID Anfrage an den Grid- Checkpointer senden bevor sie mit ihrer eigentlichen Arbeit fortfährt. Alle Abhängigkeiten werden dabei in einem pro Job-Unit eindeutig identifizierbaren Ordner festgehalten.
Eine weitere Erweiterungsmöglichkeit betrifft die Wiederherstellung einer Anwendung. Die aktuelle Implementierung ist zur Zeit in der Lage die verteilten Anwendungen wiederherzustellen, bei denen die Job-Units entweder als Client oder als Server agieren. In Abschnitt 3.5.1 wurde ein Protokoll konzeptuell erarbeitet, welches sich um die Wiederherstellung der Job-Units kümmert, die sowohl als Client, wie auch als Server fungieren. Die Implementierung dieses Protokolls stellt eine weitere Optimierungsmöglichkeit dieser Arbeit dar.
Derzeit findet die Berechnung eines global konsistenten Zustandes ausgehend vom letzten Checkpoint aller Job-Units statt. An dieser Stelle wäre es möglich die Job-Unit und die Version ab der die Berechnung gestartet werden soll, explizit anzugeben. Dadurch könnte das Verhalten der einzelnen Job-Units zu einem bestimmten Zeitpunkt reproduziert werden. Allerdings bedarf eine solche Anpassung der Modifikation des Rollback-Propagation Algorithmus, da dieser stets vom letzten Checkpoint aller Job-Units ausgeht.
Bei der aktuellen Implementierung ist der Benutzer dafür zuständig, eine Wiederherstellung einzuleiten. Dieses Verhalten könnte optimiert werden, indem das System automatisch im Fehlerfall eine Wiederherstellung der ausgefallenen Job-Units durchführt.
In Abschnitt 4.5 wurde eine Bibliothek *LibSwitch* zum adaptiven Umschalten der Check-

point Strategie eingeführt. Da vor einem Nachrichtenaustausch zunächst sichergestellt werden muss, dass die Bibliothek für unabhängiges Checkpointing bei allen beteiligten Job-Units geladen wurde, ist es zur Zeit nicht möglich zwischen dem koordinierten und unabhängigen Checkpointing umzuschalten. Dies bedarf einer Ausarbeitung und Implementierung eines zusätzlichen Protokolls, das zu dem Umschaltzeitpunkt aktiv wird und genau dies sicherstellt.
Die aktuelle Implementierung setzt lediglich eine einfache Version des unabhängigen Checkpointings ohne die Aufzeichnung der Nachrichtenkommunikation ein. Dadurch besteht die Gefahr des in Abschnitt 2.2.1 erläuterten Domino-Effekts. Eine Alternative wäre der Einsatz einer log-basierten Wiederherstellung. Dabei können die Nachrichten innerhalb der bestehenden CRUC-Bibliothek aufgezeichnet und während der Wiederherstellung in der selben Reihenfolge eingespielt werden.
Zuletzt bedarf es wohl bei der Implementierung mit als auch ohne die Aufzeichnung der Nachrichten eines Garbage-Collection Algorithmus, der sich um die Beseitigung der nutzlos gewordenen Sicherungen kümmert.

Literaturverzeichnis

[Abb88] ABBOTT, R. J.: Resourceful systems and software fault tolerance. In: *IEA/AIE '88: Proceedings of the 1st international conference on Industrial and engineering applications of artificial intelligence and expert systems*. New York, NY, USA : ACM, 1988. – ISBN 0–89791–271–3, S. 992–1000.

[BCH+03] BOUTEILLER, Aurélien; CAPPELLO, Franck; HERAULT, Thomas; KRAWEZIK, Géraud; LEMARINIER, Pierre; MAGNIETTE, Frédéric: MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In: *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 1–58113–695–1, S. 25.

[CFJ+08] CORTES, Toni; FRANKE, Carsten; JÉGOU, Yvon; KIELMANN, Thilo; LAFORENZA, Domenico; MATTHEWS, Brian; MORIN, Christine; PRIETO, Luis P.; REINEFELD, Alexander: XtreemOS: a Vision for a Grid Operating System. Version: 2008. `http://www.xtreemos.eu/publications/research-papers/xtreemos-cacm.pdf`. 2008. – Forschungsbericht.

[DG96] DAMANI, Om P.; GARG, Vijay K.: How to Recover Efficiently and Asynchronously when Optimism Fails. In: *In Proceedings of the 16th International Conference on Distributed Computing Systems*, 1996, S. 108–115.

[DHR03] DUELL, Jason; HARGROVE, Paul; ROMAN, Eric: The Design and Implementation of Berkeley Labs Linux Checkpoint/Restart. Version: 2003. `http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml`. 2003. – Forschungsbericht.

[EAWJ99] ELNOZAHY, Mootaz; ALVISI, Lorenzo; WANG, Yi-Min; JOHNSON, David B.: *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*. 1999

[FK99] FOSTER, Ian (Hrsg.); KESSELMAN, Carl (Hrsg.): *The grid: blueprint for a new computing infrastructure*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999. – ISBN 1–55860–475–8

[Gnu07] *The GNU C Library Reference Manual*. : *The GNU C Library Reference Manual*, 2007. http://www.gnu.org/software/libc/manual/pdf/libc.pdf

[JS00] JAIN, K.; SEKAR, R.: User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement, 2000, 19–34.

[Sli09] SLIWAK, Michael: *Konsistente Kommunikationskanalsicherung in Grid-Systemen*, Diplomarbeit, 2009

[SRSM08] SPAHN, John Mehnert; ROPARS, Thomas; SCHOETTNER, Michael; MORIN, Christine: The Architecture of the XtreemOS Grid Checkpointing Service / INRIA. Version: 2008. http://hal.inria.fr/inria-00346955/en/. 2008 (RR-6772). – Research Report. – 19 S.

[SS83] SCHLICHTING, Richard D.; SCHNEIDER, Fred B.: *Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems*. 1983

[Ste03] STEVENS, Richard W.: *Unix Network Programming, Vol. 1: The Sockets Networking API, Third Edition*. Addison-Wesley Professional, 2003 http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0131411551. – ISBN 0131411551

[WCLF95] WANG, Yi-Min; CHUNG, Pi-Yu; LIN, In-Jen; FUCHS, W. K.: Checkpoint Space Reclamation for Uncoordinated Checkpointing in Message-Passing Systems. In: *IEEE Transactions on Parallel and Distributed Systems* 6 (1995), Nr. 5, S. 546–554. – ISSN 1045–9219.

[WF93] WANG, Yi min; FUCHS, W. K.: Lazy Checkpoint Coordination for Bounding Rollback Propagation. In: *in Proc. IEEE Symp. Reliable Distributed Syst*, 1993, S. 78–85.

[Xtr07] XTREEMOS INRIA-TEAM: *Design and implementation of basic reconfiguration mechanisms in LinuxSSI*. XtreemOS deliverable D2.2.4, 2007

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 19. November 2009 Eugen Feller