Checkpointing in heterogenen, verteilten Umgebungen

Inaugural-Dissertation

zur
Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von
John Mehnert-Spahn
aus Elsterwerda

Düsseldorf, den 09.06.2010

Aus dem Institut für Informatik
der Heinrich-Heine Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Erstgutachter: Prof. Dr. Michael Schöttner
Zweitgutachter: Prof. Dr. Stefan Conrad
Tag der mündlichen Prüfung: 13.07.2010

Für meine Eltern.

Danksagung

Meinen herzlichsten Dank spreche ich Herrn Prof. Dr. Michael Schöttner aus, der mir die Möglichkeit zu dieser Dissertation gegeben hat. Ich bin sehr dankbar für die konsequente und großzügige Unterstützung und dem damit verbundenen persönlichen Einsatz, der entscheidend zu dieser Arbeit beigetragen hat.
Des Weiteren bedanke ich mich recht herzlich bei Herrn Prof. Dr. Stefan Conrad für die Begutachtung dieser Arbeit.

Zu herzlichem Dank verplichtet bin ich meinen Kollegen Florian Müller, Kim-Thomas Rehmann, Michael Braitmeier und Michael Sonnenfroh. Der fachliche und persönliche Austausch als auch unsere gemeinsamen Unternehmungen haben die letzten Jahre sehr bereichert.
Ich bedanke mich zudem bei Herrn Michael Sliwak, Herrn Ralph-Gordon Paul, Herrn Florian Klein und Herrn Simon Probst, die mich mit ihren Bachelor- und Masterarbeiten produktiv unterstützt haben. Besonders hervorheben möchte ich jedoch Herrn Eugen Feller, dessen intensive Mitarbeit und freundschaftliche Verbundenheit ich über die ver- gangenen Jahre sehr geschätzt habe.

Frau Katrin Spahn möchte ich ganz herzlich für Ihren überwiegend nächtlichen Einsatz des Korrekturlesens danken.

Frau Dr. Sibylle Steck bin ich sehr verbunden für die Anregung zur weiteren wissenschaftlichen Arbeit.

Meinen Eltern, Andrea und Uwe Spahn, danke ich zutiefst für die verständnisvolle und kontinuierliche Unterstützung, vor allem meiner Ausbildung, über die verschiedensten Stationen der letzten Jahre hinweg.

Kurzfassung

Wissenschaftliche und ökonomische Problemstellungen können in zunehmendem Maße nicht mehr mit den *lokal*, bei Forschungs- oder Wirtschaftunternehmen, verfügbaren Ressourcen bewältigt werden. Um auf dem Markt zu bestehen, müssen bestimmte Dienstleistungen aus Kostengründen ausgelagert werden können. Für einen effizienteren Wissens- und Informationsaustausch müssen Unternehmen mit anderen *temporär kooperieren*. Beide Aspekte bedingen, dass sich Unternehmen an eine sich ständig ändernde Umgebung anpassen.

Mit Gridtechnologien hingegen sind umfangreiche, verteilte Ressourcen, die durch Zusammenschluss mehrerer Standorte entstehen, *gemeinsam nutzbar*. Gridtechnologien werden bisher von verschiedenen Grid-Middleware-Implementierungen dominiert. Als Nachteil erweist sich jedoch, dass beispielsweise Legacyanwendungen abgeändert werden müssen, um auf Middleware-Lösungen ausführbar zu sein. Das EU-Projekt XtreemOS (Juni 2006-Mai 2010) entwickelt ein Grid-Betriebssystem, in dem wichtige Gridfunktionalitäten, wie Fehlertoleranz und Virtual Organisation Management, in den Kern verlagert werden. Anwendungen, die in einem XtreemOS Grid ausgeführt werden, müssen nicht abgeändert werden, um von diesen Funktionalitäten zu profitieren.

Mit einer steigenden Anzahl von Rechenknoten erhöht sich gleichzeitig die Wahrscheinlichkeit von Knotenausfällen. Um Zwischenzustände, beispielsweise lang laufender Anwendungen, bei einem Rechnerausfall nicht zu verlieren, müssen Fehlertoleranzmechanismen eingesetzt werden. Fehlertoleranz kann vor allem durch Checkpoint/Restart erzielt werden, das heißt, Anwendungszustände werden in periodischen Abständen gesichert und können im Fehlerfall wiederhergestellt werden.

In dieser Arbeit wird eine Grid-Checkpointing-Architektur (GCA) entworfen, um Grid-Fehlertoleranz zu realisieren. Hierbei wird der Schwerpunkt darauf gelegt, Heterogenität zu unterstützen. Das heißt, es wird untersucht, inwieweit unterschiedliche, existierende, knotengebundene Checkpointer-Pakete in eine GCA integriert werden können.

Die Analyse verfügbarer Checkpointer-Pakete zeigt große Unterschiede hinsichtlich deren Fähigkeiten, Prozessressourcen wie IPC, Sockets, Dateien, et cetera sichern und wiederherstellen zu können. Dies bedingt, Anwendungen mit eingesetzten Checkpointer-Paketen abzugleichen, damit alle Prozessressourcen einer Anwendung nach einem Ausfall rekonstruiert werden können. Andernfalls wird die Wiederherstellung eines konsistenten Anwendungszustands verhindert.

Kernelement der GCA bildet die sogenannte Uniforme Checkpointer-Schnittstelle (UCS), welche als einheitliche Schnittstelle zu heterogenen Checkpointer-Paketen dient. Die Schnittstellenimplementierung muss Bezug zu existierenden Checkpointing-Protokollen nehmen, dabei auf die Abbildung von Semantiken der Grid- auf jene der Gridknotenebene (Prozess- und Benutzer-Management) achten, individuelle Aufrufsemantiken beachten und Check-

pointdatei-management betreiben.

Heterogenität zeichnet sich auch in Bezug auf die Beziehung von Checkpointer-Paket und des von ihm unterstützten Containers aus. Container ermöglichen unter anderem leichtgewichtige Virtualisierung. Damit können Ressourcenkonflikte beim Restart vermieden werden, insofern Kennungen von Prozessen, IPC-Objekten et cetera bereits vergeben sind. Zum Zeitpunkt dieser Arbeit werden verschiedene und weit verbreitete Checkpointer-Pakete angepasst, um von existierenden Containertechnologien zu profitieren. In dieser Arbeit wird dargelegt, welche Auswirkungen auf die GCA entstehen, wenn verschiedene Container unterstützende Checkpointer-Pakete integriert werden.

Eine zentrale Herausforderung bei Grid-Fehlertoleranz besteht darin, verteilte Anwendungen auf Basis heterogener Checkpointer-Pakete zu sichern. Hierzu müssen *involvierte Checkpointer-Pakete miteinander kooperieren*. Kooperation darf jedoch nicht auf Kosten der Checkpoint- oder Anwendungs-Modifizierung stattfinden. Techniken wie Callbacks und Library Interposition stellen hierbei geeignete Mittel dar. Die durch die GCA erzielte Kooperation einzelner *Knoten-gebundener* Checkpointers resultiert in der *Aufwertung* letzterer, weil jeder in die Lage versetzt wird, die Sicherung und Wiederherstellung einer *verteilten* Anwendung zu unterstützen.

Obwohl in der Literatur ein umfangreiches Wissen über verschiedene Checkpointingprotokolle existiert, implementieren die wenigen verteilten Checkpointer-Pakete ausschließlich koordiniertes Checkpointing. MPI-Umgebungen bilden hierbei eine Ausnahme, jedoch stellen MPI-Anwendungen nur einen Bruchteil aller Anwendungen dar. Die GCA bietet eine Plattform, um in anwendungstransparenter Weise mehrere Checkpointingprotokolle zu realisieren, was eingesetzt werden kann, um Checkpointingperformanz zu erhöhen.

Das Ziel adaptiven Checkpointings ist, Checkpointingperformanz zu erhöhen. Zur Bestimmung der geeigneten Checkpointingstrategie wird das Anwendungs- und Systemverhalten einbezogen. In dem Zusammenhang müssen beispielsweise der Netzwerkverkehr oder das Schreibzugriffsmusters einer Anwendung beobachtet werden. Um einen Effizienzgewinn zu erzielen, muss der durch die Überwachung entstehende Aufwand gering gehalten werden.

Anhand dieser Arbeit wird gezeigt, dass Grid-Fehlertoleranz mithilfe heterogener Checkpointer-Pakete realisierbar ist. Der entwickelte Prototyp integriert drei verschiedene Checkpointer-Pakete, welche verwendet werden, um Anwendungen koordiniert, unabhängig und inkrementell zu sichern und wiederherzustellen. Die Anwendungen müssen dabei nicht abgeändert werden.

Die umfangreichen Messungen belegen, dass die GCA keinen nennenswerten Aufwand gegenüber nativen Checkpointer-Paketen verursacht. Mit einem wachsenden Ressourcenumfang steigt der Checkpointing- und insbesondere der Restart-Aufwand für verteilte Anwendungen, was überwiegend auf die Eigenschaften zugrunde liegender Netzwerk- und Speichertechnologien zurückzuführen ist.

Inhaltsverzeichnis

1 Stand der Technik

Im ersten Kapitel wird ein Überblick über relevante Themen der Fehlertoleranz aus theoretischer und praktischer Sicht gegeben. Eine grundlegende Frage hierbei lautet: *welchen potentiellen Fehlern* kann mit *welcher Technik* begegnet werden, um Fehlertoleranz zu erreichen? Hierzu werden Begriffe wie Fehler, Fehlermodell und Fehlermaskierung definiert. Anschließend werden Ausprägungen und Eigenschaften verschiedener Checkpointerwerkzeuge beleuchtet, welche bei der Konzeption einer Grid-Checkpointing-Architektur wichtig sind.
Am Ende dieses Kapitels werden die Ziele sowie der Aufbau dieser Arbeit erläutert.

1.1 Fehlertoleranz

1.1.1 Fehlerbegriff und Fehlermodelle

Anhand eines Beispiels werden die Zusammenhänge zwischen Störung, Fehler und Ausfall im Fehlertoleranzkontext verdeutlicht. Die IP-Adresse eines Rechnerknotens wird aufgelöst, indem eine Domain Name Service (DNS)-Anfrage gestellt wird. Fällt bei der zugehörigen Datenübertragung ein Bit um, werden die empfangenen Daten nicht im Sinne des DNS-Servers interpretiert, sodass eine abweichende IP-Adresse übergeben wird. Die *gestörte* Datenübertragung verursacht einen IP-Adressenauflösungs*fehler*, sodass nicht mit anderen Anwendungskomponenten kommuniziert werden kann und demzufolge ein Anwendungs*ausfall* entsteht.

Fehlermodelle werden nach *gutmütigen und bösartigen* Fehler eingeteilt, siehe Abbildung 1.1. Die Bezeichnung gutmütig basiert auf der verhältnismäßig einfachen Identifizierung des zugrundeliegenden Fehlers. Demnach führt ein fehlerhafter Prozess keine Aktionen durch, die ein korrekt arbeitender Prozess niemals ausführen würde. Zu gutmütigen Fehlern zählt der sogenannte Crash Failure. Ein System oder eine Anwendung stürzt ab und reagiert anschließend nicht mehr. Dieses Verhalten wird als *Fail-Stop* bezeichnet [127]. Zur Klasse der gutartigen Fehler zählen ebenfalls das Omission Failure-Modell, womit Dienstverweigerungen assoziiert werden, und das Timing Failure-Modell, welches Zeitfehler beinhaltet.

Die Klasse der bösartigen Fehler subsumiert Fehler, die schwer identifizierbar sind. Hierbei können fehlerhafte Prozesse beliebige, von korrekt arbeitenden Prozessen abweichende, Aktionen durchführen. Zusätzlich können fehlerhafte Prozesse miteinander lokal, beziehungs-

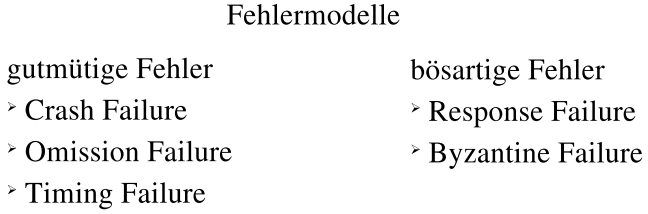

Abbildung 1.1: Fehlermodelle

weise über Knotengrenzen hinweg miteinander kooperieren und somit Dritte täuschen oder gezielt stören, siehe Antwortfehler (engl. Response Failure) und byzantinische Fehler (engl. Byzantine Failure). Ein fehlertolerantes System maskiert gutartige und bösartige Fehler. In der Praxis ist das jedoch entweder gar nicht oder nur mit sehr großem Aufwand möglich.

Definition 1: Fehlertoleranz. Fehlertoleranz bezeichnet die Fähigkeit eines Systems, auch mit einer begrenzten Anzahl fehlerhafter Komponenten seine spezifizierte Funktion zu erfüllen [61].

In der Realität existieren Systeme, beispielsweise des Verkehrsüberwachungs- oder Gesundheitssektors, deren Ausführung von *existentieller* Bedeutung ist. Vor diesem Hintergrund wird ersichtlich, dass Fehlertoleranz in engem Bezug zu Verfügbarkeit und Zuverlässigkeit steht.

Definition 2: Zuverlässigkeit. Ein System, beziehungsweise eine Komponente ist zuverlässig, wenn dessen geforderte Funktionalität unter festgelegten Bedingungen für ein bestimmtes Zeitintervall fehlerfrei ausgeführt werden kann [61].

Definition 3: Verfügbarkeit. Verfügbarkeit hingegen bezeichnet den Grad der Betriebsfähigkeit oder Zugänglichkeit eines Systems, beziehungsweise Komponente zum Zeitpunkt der Nutzung [61].

Aufgrund der Tatsache, dass nicht alle potentiell auftretenden Fehler erkannt werden können und automatisiert behebbar sind, gibt es keine Garantie für absolute Verfügbarkeit und Zuverlässigkeit.

1.1.2 Fehlererkennung

Vor dem Hintergrund der Gutmütigkeit und Bösartigkeit von Fehlern, spielt der *Zeitpunkt der Fehler-Erkennung* eine wesentliche Rolle. Manche Fehler werden *niemals* erkannt. Für die Anderen gilt, je eher sie erkannt werden, desto geringer ist das Risiko, dass *schleichende Fehler* entstehen. Schleichende Fehler können lange unerkannt bleiben. Werden sie jedoch erkannt, muss auf einen weit zurückliegenden konsistenten Zustand zugegriffen werden. Hierbei gehen unter Umständen viele Zwischenzustände verloren.

Fehlererkennung kann mithilfe von Hardware- oder Softwarekomponenten als auch der Kombination beider, siehe [52], erzielt werden. Abhängig vom jeweiligen Fehlermodell müssen spezifische Ereignisse überwacht werden. Beim Fail-Stop-Verhalten stehen anormale Prozessterminierungen und Knotenausfälle im Vordergrund. Beide Aspekte werden in Kapitel 3.3.2.1 diskutiert.

Bei *proaktiver Fehlertoleranz* [146],[110] wird versucht, Fehler vorherzusagen, um entsprechende Massnahmen durchführen zu können, *bevor* ein Fehler eingetreten ist. Komplementär zu proaktiver existiert *reaktive Fehlertoleranz*. Hierbei wird erst dann eine Fehlererholung eingeleitet, *nachdem* ein Fehler eingetreten ist.

1.1.3 Konsistenz

Das Ziel der Fehlermaskierung besteht darin, Fehler vor anderen Komponenten eines Systems zu verbergen. Ein durch Fehler entstandener inkonsistenter Zustand soll nach einer *Fehlererkennung* im Kontext der *Fehlererholung* in einen konsistenten Zustand überführt werden.

Definition 4: Konsistenter Zustand einer sequentiellen Anwendung. Ein Anwendungszustand ist konsistent, wenn er einem möglichen, während der fehlerfreien Ausführung auftretenden, Zustand entspricht. Das heißt insbesondere, dass dieser Zustand nicht zwangsweise dem vor dem Fehler gültigen, sondern einem beliebigen, vor dem Fehler, eingetretenen Zustand entsprechen kann.

Bei verteilten Anwendungen, deren Prozesse miteinander interagieren, müssen weitere Zusammenhänge beachtet werden.

Definition 5: Global konsistenter Zustand einer verteilten Anwendung. Ein globaler Anwendungszustand ist konsistent, wenn er einem möglichen Zustand während der fehlerfreien Ausführung entspricht [45]. Im Bezug auf Nachrichtenkommunikation ist der Zustand konsistent, wenn das Sendeereignis eines sendenden Prozesses und das zugehörige Empfangsereignis des betreffenden Empfängers verzeichnet sind [29]. Da Interaktionen zwischen verteilten Prozessen auch Ressourcen wie verteilte Dateisysteme und gemeinsam genutzte Segmente einschließen, kann der Konsistenzbegriff bei verteilten Anwendungen allgemeiner beschrieben werden. Nach [44] bedeutet konsistent, dass die lokalen Komponentenzustände und die aktuellen Interaktionen mit anderen Komponenten die (Protokoll-, beziehungsweise Dienst-) Spezifikation nicht verletzen.

Hingegen entsteht ein inkonsistenter Anwendungszustand, falls eine Nachricht zwar als empfangen, jedoch nicht als abgesandt vermerkt wird, siehe Abbildung 1.2. Dieses Szenario tritt ein, wenn einzelne Zustände zu unterschiedlichen Zeitpunkten, aufgrund des Fehlens einer globalen Zeit, in einem verteilten System gesichert werden.

Die Erkennung eines global konsistenten Zustandes einer verteilten Anwendung ist für Checkpointing, welches im folgenden Abschnitt erläutert wird, relevant.

1.2 Rückwärtsbehebung

Werden Zustände des Systems und/oder der Anwendung periodisch extrahiert und auf einem *stabilen Speichermedium* abgespeichert, kann im Fehlerfall zu diesen, in der Vergangenheit liegenden, Zuständen *zurückgekehrt* werden. Diese Art der Fehlermaskierung wird daher als Rückwärtsbehebung (engl. Backward Error Recovery (BER)) bezeichnet. [1]

BER kann in *Checkpoint-basierte* und *Log-basierte* Verfahrensklassen unterteilt werden. Aufgrund der Bedeutung beider Klassen für die vorliegende Arbeit werden unkoordiniertes, koordiniertes und kommunikationsinduziertes Checkpointing, als Vertreter Checkpoint-basierter Verfahren, sowie verschiedene Log-basierte Verfahren, als auch hybride Verfahren, erläutert.

1.2.1 Abhängigkeitsauflösung und Konsistenz

Existierende BER-Verfahren unterscheiden sich unter anderem hinsichtlich des Aufwands während der fehlerfreien Ausführung und beim Sichern, sowie des Aufwands bei Wiederherstellung und der Verwaltung von Checkpointdateien. Entscheidenden Einfluss darauf hat der Aspekt, ob Prozessabhängigkeiten erkannt und berücksichtigt werden. Abhängigkeiten zwischen zu sichernden Prozessen können durch Prozessverwandtschaften (Eltern-Kind) als auch durch Nachrichtenaustausch und Verwendung gemeinsam genutzter Ressourcen, wie Dateien, Segmente, Semaphore, Message Queues, Pipes oder Geräte, entstehen. Werden Abhängigkeiten nicht erkannt und berücksichtigt, können Inkonsistenzen während der Zustandssicherung als auch bei der Zustandswiederherstellung auftreten. Die vor Ausfall gültige Sicht auf gemeinsam genutzte Ressourcen darf nicht durch eine Sicherungs- und Wiederherstellungsoperation inkonsistent werden.

Im Folgenden werden die beiden charakteristischen Auswirkungen, die entstehen, wenn Abhängigkeiten nicht korrekt behandelt werden, beispielhaft an einer Nachrichten austauschenden verteilten Anwendung mit zwei Prozessen, erläutert. Abbildung 1.2 stellt das Beispiel einer sogenannten *verwaisten Nachricht (orphan message)* dar. Hierbei entspricht Checkpoint (CP) 1 P1's Zustand, eine Nachricht empfangen zu haben. CP2 entspricht P2's Zustand, keine Nachricht ausgesendet zu haben. Nach Neustart von beiden Checkpoints empfängt P1 die Nachricht erneut, was inkonsistent ist. Das heißt, die Nachricht wird an P1 erneut ausgeliefert, sodass interne Anwendungszustände entstehen, die während der fehlerfreien Ausführung nicht entstehen würden.

In Abbildung 1.3 wird das Szenario einer sogenannten *verlorenen Nachricht (engl. lost message)* dargestellt. CP2 entspricht P2's Zustand, eine Nachricht ausgesendet zu haben.

[1] Im Gegensatz zu BER existiert die Vorwärtsbehebung (engl. Forward Error Recovery (FER)). Nach [44] versetzt FER "Komponenten in einen konsistenten Zustand, ohne auf Zustandsinformationen zurückzugreifen, die in der Vergangenheit zum Zweck der Fehlertoleranz (d.h. als Rücksetzpunkt) abgespeichert wurden".

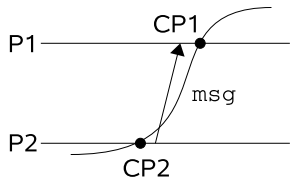

Abbildung 1.2: Verwaiste Nachricht

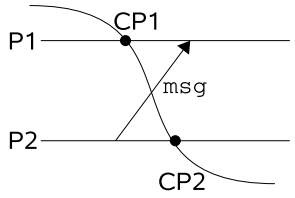

Abbildung 1.3: Verlorene Nachricht

Nach CP1 hat P1 jedoch keine Nachricht empfangen. Nach Neustart von beiden Checkpoints wird die Nachricht nicht an P1 gesendet, sie gilt damit als verloren gegangen. Auch hier entsteht ein interner Zustand, der während der fehlerfreien Ausführung nicht auftreten würde.

Beide Szenarien entstehen, wenn die Zugehörigkeit sogenannter *in-Transit Nachrichten* zu einzelnen lokalen Checkpoints eines globalen Checkpoints nicht korrekt vorgenommen wird.

Eine weitere Schwierigkeit stellt die wechselseitige Interaktion zwischen System und Außenwelt dar.

Definition 6: Außenwelt. Sie besteht aus allen Ein- und Ausgabegeräten, die nicht zurückgerollt werden können [45].

Bei einem Fehler im System kann die Außenwelt vom System nicht zurückgerollt werden, da das System keinen Einfluss auf sie besitzt. Trotzdem muss der Außenwelt ein konsistentes Systemverhalten vermittelt werden. Hierzu muss einerseits derjenige Systemzustand, von dem eine Ausgabenachricht an die Außenwelt abgesandt wurde, wiederhergestellt werden können, unabhängig von zukünftig auftretenden Fehlern. Zum sogenannten *Output Commit* Problem kommt es, wenn ein Bankautomat, als Beispiel für ein System, im Zuge einer Wiederherstellung *erneut* Geld auszahlt, weil die erstmalige Auszahlung, vor Auftreten des Fehlers, nicht im System vermerkt wurde [45]. Andererseits müssen Nachrichten von der Außenwelt an das System gesichert werden, da sie bei einer Systemwiederherstellung nicht erneut von der Außenwelt reproduziert werden können.

1.2.2 Checkpointing-basierte BER-Protokolle

Bei Checkpointing werden Anwendungszustände extrahiert und in Form von *Checkpoints* oder *Prozessabbildern* auf stabilem Speicher abgelegt. Die Checkpoints werden infolge eines Ausfalls beim Restart verwendet, um die Anwendung wiederherzustellen.

1.2.2.1 Unkoordiniertes Checkpointing

Bei diesem Protokoll besteht der größte Vorteil darin, dass ein Prozess, unabhängig von anderen Prozessen einer verteilten Anwendung, über seine Sicherung entscheiden kann.

Durch Nachrichtenkommunikation entstehende Abhängigkeiten werden während der fehlerfreien Laufzeit in Form sogenannter Abhängigkeitsinformationen aufgezeichnet. Sender-Metadaten werden im Huckepack-Verfahren der Nachricht beigefügt und zusammen mit Empfängermetadaten bei Nachrichtenauslieferung aufgezeichnet [11]. Bei Wiederherstellung fordert der Wiederherstellungsprozess P alle Abhängigkeitsinformationen mithilfe einer Anfragenachricht bei allen involvierten Prozessen an. P versucht auf Basis der empfangenen Informationen eine sogenannte Recovery Line [122] zu ermitteln, welche dem letzten konsistenten globalen Checkpoint entspricht.

Bei Erfolg sendet P jedem Prozess eine Rollbackanfragenachricht. Gehört der aktuelle Zustand des empfangenden Prozesses zur Recovery Line, wird mit der Berechnung fortgefahren. Ansonsten wird zu einem früheren Checkpoint zurückgerollt, welcher durch die Recovery Line ausgewiesen wird. Weil auf ältere Zustände zurückgerollt wird, wird diese BER Klasse auch als Rollback-Recovery Klasse bezeichnet.

Ist die Recovery-Line-Berechnung erfolglos, tritt der sogenannte *Domino-Effekt* ein, das

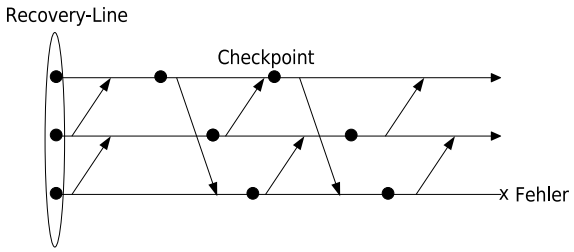

Abbildung 1.4: Recovery-Line und Dominoeffekt

heißt, es existiert kein global konsistenter Zustand, der sich aus den einzelnen lokalen Checkpoints zusammensetzt. Somit führt ein *kaskadierender Rollback* bei jedem Prozeses zurück bis zum initialen Zustand, siehe Abbildung 1.4.

Aufgrund der unabhängig erzeugten Checkpoints kann sehr viel Speicherplatz verbraucht werden, da nicht jeder lokale Checkpoint einem global konsistenten Zustand zugehörig ist.

Um jene Checkpoints zu ermitteln, die gelöscht werden dürfen, muss die Recovery Line-Berechnung durchgeführt werden, ohne gleichzeitig eine Wiederherstellung zu initiieren. Die Performanz dieses Checkpointingprotokolls ist daher an einen effizienten Speicherfreigabemechanismus gebunden. Im Vergleich zu koordiniertem Checkpointing erlaubt es eine höhere Fehlerrate, siehe [124].

1.2.2.2 Koordiniertes Checkpointing

Jedes Prozessabbild ist Teil eines global konsistenten Checkpoints einer Anwendung mit ein oder mehreren Prozessen. Das vereinfacht die Wiederherstellung, da zu diesem Zeitpunkt kein global konsistenter Zustand errechnet werden muss, der Domino-Effekt wird ausgeschlossen. Außerdem reduziert sich der Speicherbedarf gegenüber unkoordiniertem Checkpointing, weil hierbei jeder Checkpoint Teil eines global konsistenten Zustands ist.

Soll ein global konsistenter Zustand zum Checkpoint-Zeitpunkt ermittelt werden, müssen alle Anwendungsprozesse synchronisiert werden, da keine in-Transit Nachrichten ignoriert, keine gemeinsam genutzten Ressourcen mehrfach, beziehungsweise mit unterschiedlichen Inhalten gesichert werden dürfen.

In [142] wird *blockierendes Checkpointing* erläutert, welches auf einem *adaptierten*[2] Zwei-Phasen-Commit-Protokoll basiert. Während der Protokollausführung werden alle Anwendungskommunikationen blockiert. Ein ausgewiesener Koordinatorprozess sichert sich selbst und fordert anschließend in der ersten Protokollphase jeden Anwendungsprozess zur Sicherung auf. Dieser unterbricht seine eigene Ausführung, leert seine Kommunikationskanäle, nimmt einen provisorischen Checkpoint und bestätigt anschließend dem Koordinator, dass diese Aktionen beendet wurden. Nach dem Erhalt aller Prozessbestätigungen initiiert der Koordinator die Commit-Phase. Hierbei transformiert jeder Anwendungsprozess den provisorischen in einen permanenten Checkpoint und fährt danach mit seiner Ausführung fort.

Der mit blockierendem Checkpointing assoziierte Overhead, welcher durch Prozesssynchronisierung entsteht, kann vermieden werden. In [29] wird zur Anwendungsausführung nebenläufiges Checkpointing beschrieben. Der Checkpoint-Impuls wird durch Propagierung von Marker-Nachrichten bei jedem Anwendungsprozess initiiert, ohne dass es einer, von einem zentralen Koordinator überwachte, Synchronisierung bedarf.

1.2.2.3 Kommunikationsinduziertes Checkpointing

Dieses Protokoll verwendet jene Synergien, die entstehen, wenn Eigenschaften koordinierten und unkoordinierten Checkpointings kombiniert werden. Hierbei entscheiden Prozesse

[2] Ursprünglich werden Zwei-Phasen-Commit-Protokolle eingesetzt, um Konsistenz in verteilten Datenbanken zu erzielen [66]

frei über deren Sicherung, der Dominoeffekt wird jedoch vermieden, [26].

Wird unabhängig gesichert, muss darauf geachtet werden, dass sich die Recovery Line weiterentwickelt. Daher werden zusätzlich zu unabhängigen Checkpoints, *erzwungene Checkpoints* generiert. Letztere werden ausgelöst, indem ein Sender zusätzliche Informationen einer Nachricht im Huckepack-Verfahren hinzufügt, welche den Empfänger veranlassen, einen Checkpoint zu erzeugen.

Mit zunehmender Prozessanzahl skaliert dieses Protokoll jedoch nicht ausreichend, siehe [13], und wird demnach in dieser Arbeit nicht weiter betrachtet.

1.2.3 Log-basierte BER-Protokolle

Diese Verfahrensklasse kombiniert Checkpointing mit dem Aufzeichnen sogenannter *Determinanten*.

Definition 7: Determinante. Eine Determinante enthält Informationen über ein nichtdeterministisches Ereignis, wie beispielsweise einen Nachrichtenempfang, welche in Form von Tupeln kodiert werden.

Im Fehlerfall können mithilfe des letzten Checkpoints und der aufgezeichneten Determinanten *jüngere Zustände* wiederhergestellt werden, als mit den in Kapiteln 1.2.2.2, 1.2.2.3 und 3.4.4.3 vorgestellten Checkpointingprotokollen. Die Wiederherstellung ist immer *identisch* mit der Systemausführung bis zum Eintritt eines Fehlers. Nichtdeterminismus[3] wird daher bei der Wiederherstellung ausgeschlossen. Log-basierte BER-Protokolle sind wichtig für Anwendungen mit häufigen Interaktionen zur Außenwelt, vor allem jene mit nicht sicherbaren Ein- und Ausgabegeräten.

Voraussetzung für Log-basiertes Recovery ist die Modellierbarkeit der Prozessausführung als Abfolge deterministischer Zustandsintervalle, wobei hierbei jedes Intervall mit einem nichtdeterministischen Ereignis, wie einem Nachrichtenempfang oder einem Prozess-internen Ereignis, beginnt. Log-basiertes Rollback Recovery Verfahren unterscheiden sich unter anderem darin, welche Ereignisse von der stückweise deterministischen Annahme abgedeckt werden können [45]. Hierbei stehen Systemaufrufe und asynchrone Signale im Vordergrund. Aus Konsistenzgründen darf beispielsweise nicht jeder Systemaufruf bei Wiedereinspielung nicht-deterministischer Ereignisse erneut ausgeführt werden, beispielsweise Zeit- oder Datumsabfragen. Bestimmte System-Aufrufe bedürfen spezieller Betriebssystem-Unterstützung, um die gleichen Werte wie während der fehlerfreien Ausführung zurückzugeben, beispielsweise Namensräume zur Virtualisierung von Ressourcen-Bezeichnern. Asynchrone Signale in Form von Software-Interrupts müssen bei der Wiedereinspielung erneut erzeugt werden, was zu einem Checkpoint nach jedem Signal führen kann und damit teuer ist. Die Wiedereinspielung von Veränderungen gemeinsam genutzter Speicherbereiche

[3] Nichtdeterminismus bezieht sich hier auf unterschiedliches Verhalten des gleichen Programms bei mehrfacher Ausführung, welches auf eine variierende Reihenfolge elementarer Ausführungseinheiten basiert. Letzteres ist auf Schedulerentscheidungen zurückzuführen.

durch Multithread-Anwendungen stellt ebenfalls eine große Herausforderung dar, dessen Lösung im Spannungsfeld zwischen Transparenz, Effizienz und Hardware-Heterogenität liegt.

Log-basierte BER garantiert, dass *nach Wiederherstellung* aller fehlerhaften Prozesse kein sogenannter *Orphan-Prozess* vorhanden ist [45].

Definition 8: Orphan-Prozess. Dies entspricht einem Prozess, dessen Zustand von einem nicht wiederherstellbaren nicht-deterministischen Ereignis abhängt.

Abhängig davon, wie Orphan-Prozesse behandelt werden, existiert optimistisches, pessimistisches und kausales Log-basiertes Rollback Recovery.

1.2.3.1 Optimistisches Logging

Bei *optimistischem Logging* werden unter der Annahme, dass Logging vor einem Fehler erfolgreich beendet wurde, Determinanten *asynchron*, das heißt, vorerst in einem flüchtigen Speicher und später auf stabilem Speicher, abgelegt [136]. Der Aufwand im fehlerfreien Betrieb reduziert sich dadurch. Bei Verlust der im flüchtigen Speicher befindlichen Determinanten kann ein Empfänger zum *Orphan-Prozess* werden, wenn ein fehlerhafter Sender eine Nachricht verschickt, die jedoch infolge des Verlustes nicht wiederhergestellt werden kann. Der Sender muss dann zurückgerollt werden, was die Komplexität und den Zeitaufwand des Wiederherstellungsprozesses erhöht.

1.2.3.2 Pessimistisches Logging

Bei *pessimistischem Logging* wird in *synchroner* Weise die Determinante jedes nicht-deterministischen Ereignisses vor der Auswirkung auf die Anwendungsberechnung aufgezeichnet, sodass niemals ein Orphan-Prozess auftreten kann. Nach [2] besitzen pessimistisches und optimistisches Logging vergleichbare Performanzdaten.

Bei tiefergehender Unterteilung des pessimistischen Verfahrens kann in Abhängigkeit der Seite, auf der die Determinanten aufgezeichnet werden in *Sender-basiertes* und *Empfänger-basiertes* pessimistisches Log-basiertes Rollback Recovery unterschieden werden. Bei letzterem muss der wiederherzustellende Prozess nicht mit jenen synchronisiert werden, die nicht von einem Fehler beeinflusst wurden. Hierdurch wird eine höhere Skalierbarkeit erzielt [42].

1.2.3.3 Kausales Logging

Kausales Logging strebt an, geringe Leistungseinbußen [10] und ein verbessertes Ausgabeverhalten miteinander zu kombinieren. Unter Umständen bedarf es jedoch einer umfangreichen Wiederherstellung und Speicherfreigabe. Orphan-Prozesse werden vermieden,

da ein Sender nicht-stabile Determinanten der Nachricht hinzufügt und diese auf der *Empfängerseite* im flüchtigen Speicher abgelegt werden.

1.2.4 Hybride Verfahren

Bestehende Checkpointing- und Logging-Protokolle können miteinander kombiniert werden, um Synergien zu erzeugen und deren individuelle Nachteile zu eliminieren, beziehungsweise zu reduzieren.

Wird Log-basierte Wiederherstellung mit unkoordiniertem Checkpointing kombiniert, wird der Domino-Effekt ausgeschlossen. Wird koordiniertes Checkpointing mit Sender-basiertem Logging kombiniert, müssen keine Nachrichten-Logs des flüchtigen Speichers auf stabilen Speicher geschrieben und verwaltet werden, siehe [45], da sie bereits Teil des zu sichernden Adressraumes sind. Dies reduziert Aufwand und resultiert in einer einfacheren Implementierung.

Beide Mischformen werden unterschiedlich bewertet. Nach [90] ist unkoordiniertes Checkpointing mit Logging in einem großen Cluster und Auftreten *eines* Fehlers pro Stunde bei MPI-Anwendungen mit großem Datenbereich der Nutzung von koordiniertem Checkpointing mit Logging vorzuziehen. In [46] wird koordiniertes Checkpointing mit Logging, bei dem nur die Abhängigkeitsinformationen, jedoch nicht die Nachricht aufgezeichnet wird, als performanter gegenüber unkoordiniertem Checkpointing mit regulärem Logging dargestellt. Hierdurch erhöht sich die Performanz während der fehlerfreien Ausführung, es werden eine verkürzte Wiederherstellungsdauer, eine vereinfachte Checkpointdatei-Verwaltung und eine reduziertere Komplexität erreicht.

O2P-CF ist ein nachrichtenaufzeichnendes Fehlertoleranzprotokoll für Anwendungen, die auf föderierten Clustern ausgeführt werden, siehe [124]. O2P ist ein optimistisches Protokoll für Clusteranwendungen, welches mit einem Empfänger-basierten pessimistischen Protokoll zu O2P-CF, auf Cluster-zu-Cluster-Ebene, kombiniert wird. Hierdurch wird die Unabhängigkeit verschiedener Cluster gewährt, während optimistisches Logging optimale Performanz im fehlerfreien Betrieb erlaubt.

1.2.5 Replikation und zuverlässige Gruppenkommunikation

Datenbanken, Rechenprozesse und Dienste werden repliziert, um die Verfügbarkeit und die Zuverlässigkeit zu erhöhen[4]. Es existieren zwei grundlegende Replikationstechniken. Bei *aktiver Replikation* werden Lese- und Schreibanfragen in dezentraler Weise an *alle* Repliken geleitet und verarbeitet. Bei *passiver Replikation* werden Anfragen durch *ein ausgewiesenes Replikat* verarbeitet und Zustandsänderungen werden anschließend an alle anderen

[4]Replikations-basierte Fehlertoleranz bietet jedoch bei Ausfall *aller* Knoten keine Möglichkeit, Zustände wiederherzustellen, da zuvor keine Zustände gesichert worden sind.

geleitet.

Nach [67] ist die Implementierung von Gruppen-Multicastprimitiven die zentrale Herausforderung bei Fehlertoleranz aufbauend auf Replikation. Dem *Linearisierbarkeits-Kriterium* zufolge müssen alle Replikate hinsichtlich der auszuführenden Operationen und deren Ausführungsreihenfolge übereinstimmen. Während atomarer Multicast mit totaler Nachrichtenordnung letzteres erfüllt, kann Gruppendynamik, das heißt, wegfallende oder zur Gruppe hinzutretende Prozesse, mit dieser Technik nicht erzielt werden. Erst Ansätze wie die virtuelle Synchronität [22] ermöglichen eine zuverlässige, atomare Nachrichtenauslieferung an dynamische Prozessgruppen.

Virtual Nodes [38] replizieren zustandsgebundene, interaktive Java-basierte Anwendungsdienste im Grid, um den potentiellen Ausfall einer oder mehrerer Instanzen zu maskieren. Hierbei werden einerseits sich ändernde Replikatsgruppen, andererseits aktive und passive Replikation unterstützt. Zusätzlich ist der Ansatz flexibel in Bezug auf Integration heterogener Middleware-Schnittstellen wie J-RMI, SOAP, CORBA, et cetera.

Virtual Nodes nutzt die Anycast-Funktionalität von Mobile IPv6. Demnach gibt eine Gruppe verteilter Replika-Server vor, ein mobiler Knoten zu sein. Ein Replika-Server ist durch Adressübersetzung auf IP-Level auch dann erreichbar, wenn er sein Heim-Netzwerk verlässt, infolge der Instabilität eines Knotens. Über einen Home Agent kann Kontakt zu der Gruppe aufgebaut werden. Da mobile Knoten miteinander kooperieren, kann der Kontakt-Knoten auch wechseln. Replika-Aktualisierungsnachrichten können damit an alle Knoten der Gruppe zugestellt werden, wenn *eine* Adresse genutzt wird.

Der Virtual Nodes-Ansatz birgt jedoch wesentliche Nachteile im Hinblick auf Multithreading-Anwendungen. Multithreading impliziert Nicht-Determinismen, das replikationsgebundene Linearisierbarkeitskriterium kann damit nicht ohne entsprechende Maßnahmen eingehalten werden. Insofern nur Ein-Thread-Anwendungen zugelassen werden, erfolgt nicht nur eine Einschränkung der unterstützten Anwendungstypen, darüber hinaus werden Multikernprozessoren nicht effizient ausgenutzt. Virtual Nodes setzt *deterministischen Multithreading* ein, wonach knotenübergreifend, in deterministischer Weise, zwischen Threads umgeschalten wird. Hierfür wird anstelle des Java Virtual Machine (JVM) Schedulers ein eigener Scheduler verwendet. Als Nachteil daran erweist sich die verminderte Nebenläufigkeit, die Notwendigkeit, die Anwendung abzuändern (Quelltext- oder Binärtext-Ebene) und zusätzliche Software-Komponenten installieren zu müssen.

In [138] wird Replikation verwendet, um *Single-Point-of-Failures* und *Single-Link-of-Failures* zu vermeiden. Dabei können gleichzeitig Performanzgewinne paralleler Algorithmen verzeichnet werden.

1.3 Klassifizierung bestehender Implementierungen

Checkpointer sind Software-Komponenten, welche diverse Anwendungs-, System und Hardware-Zustände des Prozessors, des Speichers, des Netzwerks, der Festplatte, et cetera in unterschiedlichem Umfang sichern und wiederherstellen. Im Folgenden wird unterschieden zwischen *anwendungsbasiertem* und *systembasiertem Checkpointing* sowie MPI-Umgebungen mit integrierter Fehlertoleranz.

1.3.1 Anwendungs-Checkpointing

Anwendungscheckpointer extrahieren Prozesszustände von Anwendungen, um sie in einer Betriebssystemumgebung wieder aufbauen zu können. Diese Checkpointer lassen sich hinsichtlich ihrer jeweiligen Implementierungsebene klassifizieren.

1.3.1.1 Kernel-Checkpointer

Kernel-Checkpointer sind auf Betriebssystemebene implementiert. Unter Linux existieren Checkpointer meist in Form ladbarer Kernmodule, beispielsweise VMADump [68], oder sind fest im Kern integriert, wie der aktuell in der Entwicklung befindliche Linux-native Checkpointer.

Kernel-Checkpointer erlauben Zustände *anwendungstransparent* zu extrahieren und wiederherzustellen, das heißt, ohne den Anwendungsquelltext modifizieren zu müssen. Transparentes Checkpointing impliziert jedoch auch, dass kein anwendungsinternes Wissen in die Sicherung einfließen kann. Wird jedoch mehr gesichert wird, als aus Anwendungsperspektive notwendig ist, entsteht ein unnötiger Mehraufwand.

Kernel-Checkpointer bieten die *größte Flexibilität hinsichtlich der Zustandsextraktion und -wiederherstellung*. Ein fest im Kern integrierter Checkpointer besitzt uneingeschränkten Zugriff auf alle, ein als Kernmodul implementierter Checkpointer besitzt Zugriff auf *fast alle*[5] Kernstrukturen. Im Kern können beispielsweise gezielt Prozessverwandtschaften (Eltern-Kind) oder Ressourcenbezeichner wiederhergestellt werden, was aufgrund der fehlenden Schnittstelle vom Benutzeradressraum aus nicht möglich ist.

Der Festplattenzugriff eines Kernel-Checkpointers ist zudem schneller als der eines Bibliotheks-Checkpointers, da Festplatteninhalte nicht zwischen Kern- und Benutzeradressraum kopiert werden müssen. Der mit Kernel-Checkpointern assoziierte Vorteil bedingt jedoch, dass diese ständig an den sich weiterentwickelnden Kern angepasst werden müssen.

[5] Kern-interne Symbole müssen explizit exportiert werden, bevor sie in Kernmodulen verwendet werden können.

1.3.1.2 Bibliotheks-Checkpointer

Checkpointing kann mithilfe von Bibliotheken auf der Benutzeradressraumebene vorgenommen werden. Wird eine Bibliothek eingebunden, sodass der Anwendungsquelltext modifiziert und neukompiliert werden muss, wird Anwendungstransparenz verletzt. Ein Bibliotheks-Checkpointer kann jedoch auch in die Anwendung eingebunden werden, indem das Anwendungsbinary modifiziert wird. Hierbei wird eine zusätzliche ELF-Sektion eingefügt, welche den *neuen* Anwendungseintrittspunkt (main-Funktion) darstellt und die Bibliotheksinitialisierung übernimmt [123]. Vollständig anwendungstransparentes Bibliotheks-Checkpointing kann durch Library Injection vorgenommen werden. Hierzu werden Standard-Bibliotheks-Aufrufe abgefangen und an die Checkpointerbibliothek weitergeleitet.

Aufgrund bestehender *standardisierter* System-Schnittstellen (POSIX) und Bibliotheken (Standard C) kann auf die Kern-Funktionalität nur in begrenztem Maße zugegriffen werden[6]. Anhand dieser Schnittstellen gelingt es jedoch, weitestgehend unabhängig von der Systementwicklung zu bleiben. In dem Zusammenhang sind Abbilder von Bibliothekscheckpointern *portabler* als jene von Kernel-Checkpointern.

Anwendungs- als auch Bibliotheks-basiertes Checkpointing kann mithilfe des sogenannten *COW (Copy-On-Write) Checkpointing*, beziehungsweise *forked checkpointing* realisiert werden. Wenn ein Eltern-Prozess P ein Kind-Prozess C erzeugt, werden anhand des Seitenschreibschutz-Mechanismus' Seitenkopien bei einem unerlaubten Schreibzugriff erstellt. Hierbei rechnet Prozess P weiter, während C die alten, unveränderten Seiten P's behält und diese gesichert werden können. Dadurch wird fast die gesamte Zeit, die beim Sichern auf Festplatte anfällt, vor P verborgen [15]. Die Prozesswiederherstellung erfolgt meist anhand eines separaten Restart-Prozesses, welcher wiederhergestellte Ressourcen an seine Kind-Prozesse vererbt.

1.3.1.3 Hybride Versionen

Hybride Checkpointer bestehen aus Bibliotheks- und Kern-Komponenten, wie besipielsweise BLCR. Der Bibliotheks-Teil dient meist zur Registrierung von Callbacks, die vor und nach dem Checkpoint-Zeitpunkt als auch nach der Wiederherstellung ausgeführt werden können.

[6] Zusätzlich zu Bibliotheksaufrufen können über das /proc-Dateisystem im Kern verfügbare Anwendungsinformationen, beispielsweise das Speicher-Layout oder existierende Dateideskriptoren, für eine konsistente Sicherung ermittelt werden.

1.3.1.4 Anwendungs-gesteuertes Checkpointing

Die Anwendung wird intern um anwendungsspezifische Checkpointing-Funktionalität erweitert [27]. In Bezug auf viele Anwendungen kann dieses Vorgehen sehr aufwendig sein, da die Wiederverwendbarkeit des Checkpointingquelltextes beschränkt ist. Der Sicherungsaufwand wird jedoch entscheidend verringert, da anwendungsinternes Wissen einbezogen wird.

Checkpointingquelltext zu entwickeln stellt hohe Anforderungen an Anwendungsprogrammierer, da Betriebssystemkenntnisse integriert werden müssen. Da die Mehrheit der Entwickler diesen Anforderungen jedoch nicht nachkommen kann, ist diese Art der Fehlertoleranzrealisierung schwierig und vor allem fehleranfällig.

1.3.1.5 Verteilte Anwendungs-Checkpointer

Im Gegensatz zu Knoten-gebundenen Checkpointern existieren nur wenige Implementierungen, um verteilte Anwendungen zu sichern und wiederherzustellen. Hierzu zählen beispielsweise DMTCP [16] oder DCR [96]. In LinuxSSI, einem Single System Image Cluster-Betriebssystem [107], können im Cluster verteilte Prozesse gesichert und wiederhergesellt werden [51]. Alle Vertreter verwenden ausschließlich koordiniertes Checkpointing.

1.3.1.6 Unterschiede zwischen Anwendungscheckpointern

Die existierenden Anwendungscheckpointer variieren dahingehend, verschiedene Software-Ressourcen, wie Thread- und Prozessgruppen, IPC-Objeckte, Signal-Strukturen, Dateien, Deskriptoren, et cetera sichern und wiederherstellen zu können. Tabelle 1.1 listet die Eigenschaften funktional höherwertiger Checkpointer auf. Daraus wird ersichtlich, dass Thread- und Prozessgruppen-Strukturen[7] und Signalstrukturen meist unterstützt werden, Interprozess-Kommunikations-Techniken hingegen weiterer Entwicklung bedürfen.

Checkpointer bieten im Allgemeinen eine reduzierte Dateifehlertoleranz an, da es sehr unterschiedliche Dateisysteme gibt. Zu Pseudodateisystemen zählen /devfs, /procfs, /sysfs, /shmfs, et cetera.[8] Fehlertoleranz wird hierbei erzielt, indem beispielsweise Major- und Minornummern, Inodes und insbesondere die Kacheln, welche diese Hauptspeicherdateien beinhalten, gespeichert und wiederhergestellt werden müssen. Bei regulären Dateien, wie Text- oder Binärdateien, die auf Platte abgelegt werden, müssen Rechte, Benutzerkennungen, Inodes, Dateinamen, aktuelle Leseposition, Größe und die Festplattenblöcke, et cetera fehlertolerant verwaltet werden. Das Dateisystem btrfs [1] realisiert Fehlertoleranz intern,

[7] Hintergründe zu Thread- und Prozessgruppen werden in Kapitel 4 ausführlich behandelt.

[8] Das /devfs enthält beispielsweise Block- und Character-Gerätedateien, /procfs und /sysfs stellen eine Schnittstelle dar, um Daten zwischen Kern und Benutzerebene auszutauschen und Kernparameter einzustellen, /shmfs enthält die Dateien, welche Inhalte gemeinsam genutzter Segmente darstellen.

Eigenschaft/ Checkpointer	BLCR	MTCP	LinuxSSI	OpenVZ	Zap	Meta-Cluster
Ebene	Kern	Benutzer	Kern	Kern	Kern	Kern
Komponenten	Kern Mod. Bibl. Shell	Bibl. Shell	Kern Mod. Kern Patch Bibl. Shell	Kern Patch Container Template Shell	Kern Mod. Bibl.	Kern Mod. Bibl. Shell
Ein-Thread-Proz.	ja	ja	ja	ja	ja	ja
Mehr-Thread-Proz.	ja	ja	ja	ja	ja	ja
einf. Prozess-Baum	ja	ja	ja[a]	ja	ja	ja
UNIX-Proz.-Gruppe	ja	-	ja[b]	ja	ja	ja
UNIX-Session	ja	nein	ja	ja	-	ja
Signal	ja	ja	ja	ja	ja	ja
SYSV IPC Shmem.	nein	ja	ja	ja	ja	ja
SYSV IPC Sem.	nein	-	nein	-	ja	ja
SYSV IPC Msg.	nein	-	nein	-	-	nein
Sockets	nein	ja	nein	ja[c]	ja	ja
Pipes	ja	ja	ja	ja	ja	ja
Reguläre Dateien	geplant	nein	nein	-	-	ja
Pseudodateien	/dev/null /dev/zero	-	nein	-	ja	-
mmap Dateien	ja	ja	ja	ja	ja	ja
Inkrementell	nein	nein	ja	nein	nein	nein
Container	nein	nein	cgroups[d]	OpenVZ Container	pods	virtual bubble
Callbacks	ja	nein	ja[e]	nein	nein	ja
MPI[f]	ja	ja	nein	-	nein	ja[g]
Framebuffer	nein	nein	nein	nein	nein	nein

[a] Ein Prozess-Baum muss in einer separaten UNIX-Session gekapselt sein.
[b] Eine UNIX-Prozessgruppe muss in einer separaten UNIX-Session gekapselt sein.
[c] Basiert auf TCP-Retransmission.
[d] Derzeit existiert keine Unterstützung für Subsysteme.
[e] Es existiert keine Unterstützung für kritische Abschnitte.
[f] MPI-Implementierungen verwenden Anwendungs-Checkpointer pro Knoten.
[g] In Verbindung mit der Scali MPI Implementierung.

Tabelle 1.1: Checkpointereigenschaften

indem Dateicheckpoints und -prüfsummen eingesetzt werden.

Sogenannte *Container*, virtuelle Maschinen auf Systemaufrufsebene, isolieren Anwendungen *effizienter* voneinander als hypervisorbasierte virtuelle Maschinen, [130]. Wie aus Tabelle 1.1 ersichtlich ist, sind derzeitige Anwendungscheckpointer an individuelle Container gebunden. Aus funktioneller Sicht betrachtet, stellen Container und Checkpointer jedoch logisch voneinander getrennte Komponenten dar. Eine bidirektionale Abbildung muss zukünftig keinen weiteren Bestand haben.[9] Existierende Containertechnologien werden detailliert in Kapitel 4.3.2 beschrieben.

Checkpoint- und Restart-Callbacks werden derzeit ausschließlich von BLCR, LinuxSSI und MetaCluster unterstützt. Bei Einbindung von Bibliotheks-Checkpointern und Callbacks ist die Programmiersprache der Anwendung von Bedeutung. BLCR und MetaCluster unterstützen nur C/C++ Anwendungen.

Eine Anwendungssicherung kann *nebenläufig* und *sequentiell* vorgenommen werden. In [119] wird das *many-to-one* Schreibmuster, mehrere Adressraumbestandteile werden gleichzeitig in eine Datei geschrieben, realisiert, welches zu höherer Performanz des Checkpointings führt. In [91] läuft der Sicherungsalgorithmus größtenteils nebenläufig zu einer parallelen Anwendung und unterbricht letztere nur für kurze Momente.

Einige Checkpointer nehmen unmittelbar nach der Zustands-Extrahierung und vor der persistenten Sicherung eine Datenkomprimierung vor, sodass Abbilder einen geringeren Speicherplatzbedarf haben. Beispielhaft hierfür ist MTCP, siehe [16].

1.3.2 System-Checkpointing

1.3.2.1 System-Virtualisierung

Hardware-Virtualisierung[10] und Para-Virtualisierung ermöglichen, dass *mehrere* Betriebssysteme auf *einer einzigen* Hardware-Plattform *nebenläufig* und isoliert voneinander ausgeführt werden können. Beide Virtualisierungstechniken werden mithilfe einer Software-Komponente, dem Virtual Machine Monitor (VMM), beziehungsweise dem Hypervisor, realisiert. Jedes virtualisierte Betriebssystem wird als Domain bezeichnet und wird als Benutzer-Prozess und damit unprivilegiert oberhalb des VMMs, ausgeführt. Bei XEN wird jeder Benutzer-Domain (DomU) Zugriff auf Hardware, wie Netzwerk und Festplatte, über eine privilegierte[11] Domain (*Dom0*) erteilt, die sich auf gleicher Ebene wie die DomU's befindet. Die Speicherverteilung an die Dom0 und die DomU's hingegen obliegt ausschließ-

[9] Das cgroup Container-Framework wurde erfolgreich in Linux Mainline integriert. Der LinuxSSI Checkpointer bietet bereits Unterstützung hierfür, BLCR wird in absehbarer Zeit dahingehend portiert.

[10] Auch bekannt als Full Virtualisation.

[11] Exklusiver Zugriff auf die CPU wird in der x86 Architektur anhand von vier Prioritätsstufen geregelt, die durch vier konzentrische Ringe symbolisiert werden. Der Kern besitzt die höchste Priorität (Ring 0), Anwendungen werden im Ring 3 ausgeführt.

lich dem VMM.

Bei Hardware-Virtualisierung benötigen einige CPU-Instruktionen des Gast-Betriebssystems höhere Privilegien und können nicht direkt ausgeführt werden. Es ist die Aufgabe des VMMs, diese Instruktionen abzufangen und sie zur Laufzeit, ohne das Gast-Betriebssystem modifizieren zu müssen, sicher auszuführen [3]. Bei Para-Virtualisierung wird das Gast-Betriebssystem modifiziert, um den VMM zu unterstützen. Unnötige privilegierte Instruktionen müssen vermieden werden, da jede Seitentabellen-Modifizierung des Gast-Betriebssystems den VMM aufruft und damit teuer sind. Die mit einer VMM assoziierten Kosten sind vornehmlich auf den Ausführungswechsel zwischen Domain und VMM, beziehungsweise den Ein- und Austritt von VMM-Operationen, zurückzuführen. In [41] werden Virtualisierungkosten bei Intel Prozessoren mit 17 Prozent und bei AMD Prozessoren mit 38 Prozent beziffert.

Die mit Virtualisierung verbundenen Kosten liegen einerseits in der Notwendigkeit begründet, den Translation Lookaside Buffer (TLB), welcher die aufwendige Auflösung virtueller auf physikalische Adressen zwischenspeichert, bei VMM Ein- und Austritt zu leeren. Dadurch treten vermehrt TLB-Misses auf. Werden hingegen *TLB Inhalte, die bei einem Kontextwechsel nicht gelöscht werden sollen, markiert (TLB tagging),* kann der Umfang an TLB-Misses reduziert werden. Da diese Technik zu einer stärkeren Auslastung des Caches (engl. cache pressure) führen kann, müssen zusätzlich TLB-Einträge reduziert werden, um die Performanz zu erhöhen [41]. Letzteres wird erzielt, indem virtueller Speicher so kompakt wie möglich alloziert wird.

Andererseits kann die DomU-Speicherverwaltung zu unterschiedlichen Leistungseinbußen führen, je nachdem ob Schattenseitentabellen oder virtuellen physikalischen Speicher unterstützende Prozessoren eingesetzt werden. Unabhängig davon, ob ein paravirtualisierter Kernel oder Hardware-unterstützte Virtualisierung vorliegt, muss die Schattenseitentabelle des VMM's mit der separaten Seitentabelle des Gastbetriebssystems, wenn letztere modifiziert wurde, synchronisiert werden [40]. Bei Prozessoren, die virtuellen physikalischen Speicher unterstützen, modifiziert das Gastbetriebssystem *direkt* die Seitentabelle, die vom Prozessor verwendet wird. Letzteres führt zu weniger Speicherverbrauch und effizienterer Speicherbehandlung.

1.3.2.2 Virtual Machine-Checkpointing

System-Checkpointing kann durch Virtual Machine-Checkpointing realisiert werden. Im Gegensatz zu Anwendungscheckpointern müssen zusätzlich System-Laufzeitstrukturen, wie die Prozessor-gebundene globale Deskriptortabelle (GDT), globale Interrupt-Deskriptor-Tabelle (IDT), Softirq- und Tasklet-Arrays, die Workqueue-Liste, et cetera, Systemspeicher (verschiedene Caches), das Dateisystem, diverse Hauptspeicherdateisysteme, einschließlich von Gerätezuständen, und vieles mehr in die Sicherung einbezogen werden.

In [110] wird XEN, ein Para-Virtualisierung-realisierender Virtual Machine Monitor (VMM),

für Fehlertoleranz eingesetzt. Auch Hardware-Virtualisierung, in Form des VMMs VMWare Player oder KVM, kann hierzu verwendet werden [80].

1.3.3 Fehlertoleranz in standardisierten Programmier-Umgebungen

Parallele Anwendungen können aufbauend auf dem Nachrichten-Übertragungs-Paradigma realisiert werden. MPI (Message Passing Interface) und PVM (Parallel Virtual Machine) stellen die in diesem Bereich vorherrschenden Standards dar, wobei MPI in den letzten Jahren größere Bedeutung erhalten hat [42].

Implementierungen des MPI-Standards entlasten den Benutzer bei der Entwicklung komplexer, paralleler Anwendungen, indem von Details der Netzwerk-Implementierung, beispielsweise für Gruppenkommunikation, abstrahiert werden kann. MPI folgt dem SPMD (Single Process Multiple Data) Prinzip, n Prozesse werden auf n Prozessoren aufgeteilt und können demnach parallel ausgeführt werden. Je mehr Knoten involviert sind, desto größer ist die Gefahr eines Knotenausfalls, beziehungsweise eines MPI-Anwendungs-Absturzes. Seit einigen Jahren wird dieser Tatsache entgegengewirkt, indem Fehlertoleranz in verschiedene MPI-Umgebungen, in Form verschiedener Rollback-Recovery-Protokolle, integriert wird.

CoCheck [131] ist einer der ersten Versuche, Fehlertoleranz in MPI zu integrieren. Das Sichern der gesamten Anwendung, mithilfe der Condor Bibliothek, führte jedoch zu hohen Kosten. StarFish, [6] verwendet Protokolle atomarer Gruppenkommunikation, um das Protokoll, welches Nachrichten eines Kommunikationskanals sichert, von CoCheck zu vermeiden.

In LA-MPI [65] wird Fehlertoleranz auf Datenintegrität bezogen und mit Prüfsummenberechnung und gegebenenfalls Neuübertragung realisiert.

LAM/MPI [125] setzt BLCR zur Prozess-Sicherung ein. Kanäle werden während des Checkpointings geleert und beim Restart als leere Kanäle wiederhergestellt.

MPICH-Vx [24] basiert auf MPICH und einer Ausführungsumgebung. Fehlertoleranz wird mittels unkoordinierten Checkpointings erzielt, um zentrale Kontrolle und globale Schnappschüsse zu verhindern. Checkpointer-Server sichern Kommunikations-Kontexte und Berechnungszustände unabhängig.

MPICH2 [87] ist eine Umgebung, um speziell *dynamische* Prozesse einer MPI-Welt zu verwalten. Damit kann insbesondere Prozess-Migration, neben Fehlertoleranz, realisiert werden.

Die erwähnten MPI-Umgebungen ermöglichen vollständige (MPICH-V2, MPICH2, LA-MPI, LAM/MPI), beziehungsweise partielle (CoCheck, StarFish) Anwendungstransparenz. Zudem erlauben MPICH2- und MPICH-V2-Umgebungen, dass eine verteilte Anwendung aus sehr vielen Prozessen bestehen kann. Lediglich MPICH2 unterstützt Prozess-Migration. Keine dieser Umgebungen ist jedoch dezentralisiert ausgerichtet (Erstellung einer MPI-

Welt von einem Knoten aus unter Angabe der MPI-Knoten-Adressen) oder unterstützt ausreichend Grid-inhärente Heterogenität.

1.4 Grid-Computing

Im folgenden Abschnitt werden elementare Dienste und deren Funktion im Umfeld des Grid Computings skizziert. Diese Dienste werden in Kapitel 1.4.7 zu einer Referenzarchitektur zusammengefasst, welche das Fundament für die in Kapitel 3.1 vorgestellte Grid-Checkpointing-Architektur bildet.

1.4.1 Überblick über verteiltes Rechnen mit Diensten

Ende des vergangenen Jahrhunderts bezeichnete der Begriff *Grid* eine *Infrastruktur für verteiltes Rechnen in den Natur- und Ingenieurswissenschaften*, [55]. In der heutigen Zeit können Rechenanforderungen anderer Bereiche, wie beispielsweise der Industrie, nicht mehr durch konventionelles verteiltes Rechnen erfüllt werden. Ökonomischer Erfolg misst sich unter anderem an der Fähigkeit, sich *an eine sich ständig ändernde Umgebung anzupassen*, zum Beispiel in Form temporärer Kooperationen mit anderen Unternehmen und der *Auslagerung bestimmter Dienstleistungen aus Kostengründen*, [54].

Grid Computing ist eine Weiterentwicklung konventionellen, verteilten Rechnens, da es auf die großflächige Nutzung gemeinsamer Ressourcen fokussiert, anstelle exklusiv einen privaten Cluster zu verwenden. Zentrale Eigenschaften stellen das flexible, sichere und koordinierte Nutzen gemeinsamer Ressourcen durch dynamische Gruppierungen von Einzelpersonen und/oder Institutionen dar. Hierzu werden neue Abstraktionen und Konzepte benötigt, um die resultierenden technischen Herausforderungen wie Heterogenität, Job- und Ressourcen-Management (Erkundung, Scheduling), Sicherheit (Authentifizierung, Authorisierung), et cetera [56] zu lösen.

Ein wesentlicher Begriff im Grid Computing Kontext ist der des *Dienstes*. Die Open Grid Services Architecture (OGSA, [54]) definiert eine offene und dienstorientierte Architektur, in der ein Dienst zentrale Bedeutung besitzt und Rechen- sowie Speicherressourcen, Netzwerke, Programme und Datenbanken bezeichnet. Diese Dienste werden mit Eigenschaften von Web Services [147] vermengt, sodass *Grid Services* entstehen. Da ein Grid Service über heterogene Systeme (mit jeweils unterschiedlichen Software-Umgebungen) hinweg genutzt werden soll, müssen entsprechende Techniken verwendet werden, um Interoperabilität zu gewährleisten. Beispielsweise besitzt ein Grid Service eine Beschreibung (WSDL, Web Service Description Language, [31]), sodass für die Ansteuerung erforderlicher Client- und Servercode generiert werden kann und gleichzeitig Kompatibilität mit höherwertigen offenen Standards, anderen Diensten und Werkzeugen gegeben ist.

Im Kontext des EU-Projekts CoreGRID (2004-2008) wird postuliert, dass, auf mittlere

beziehungsweise langfristige Sicht betrachtet, komplexe Softwareanwendungen dynamisch, aus zusammengesetzen Diensten, erstellt werden. Darauf aufbauend wird das Grid als weltweiter Cyber-Versorger wahrgenommen, welcher aus miteinander kooperierenden Diensten besteht, die in einem komplexen und gigantischen Ökosystem interagieren, [99].

Es existieren unterschiedliche Ausprägungen von Grids. Bei Middleware-basierten Ansätzen, wie dem Globus Toolkit [55] werden Griddienste zwischen dem Betriebssystem und der Anwendung platziert. In XtreemOS, einem auf Linux basierenden Grid-Betriebssystem, werden native Linux-Dienste, wie Sicherheit und Checkpointing, *um Grid-Funktionalität erweitert* [150].

Es existieren unterschiedliche Formen von Grids, wie Rechengrids, Datengrids, Kooperationgrids oder Netzwerkgrids. Im Folgenden werden wichtige Konzepte, grundlegende Technologien und Komponenten des Grid-Computings erläutert, die sich auf den allgemein verbreiteten Typ des Dienstleistungsgrids beziehen, in dem Rechenzyklen, Daten und andere Ressourcen gleichzeitig gemeinsam genutzt werden.

1.4.2 Virtuelle Organisationen

Die spezifische Herausforderung im Grid besteht in der koordinierten, gemeinschaftlichen Ressourcennutzung und Problemlösung in dynamischen, multi-institutionalen virtuellen Organisationen.

Definition 9: Virtuelle Organisation. Eine Virtuelle Organisation (VO) ist eine Menge von Einzelpersonen und/oder Institutionen, welche durch Regeln der Nutzung gemeinsamer Ressourcen, sogenannten VO-Policen (engl. VO policies), definiert wird, [56].

Ressourcennutzung durch VO-Teilnemer bezieht sich hierbei auf den direkten Zugriff auf Rechner, Software, Daten und andere Ressourcen. Die *feingranulare Kontrolle der Ressourcennutzung* ist ein besonderes VO-Merkmal und wird beispielsweise durch Peer-to-Peer Systeme nicht erreicht. Aus konzeptueller Sicht können in VOs Ressourcenquotas pro VO-Mitglied separat und dynamisch eingestellt werden. OpenVZ und cgroup-Subsysteme, siehe Kapitel 4.2.4, ermöglichen beispielsweise Zugriff auf Speicher- und Prozessorleistung individuell zu steuern. VOs werden hierdurch zu einem elementaren Bestandteil eines Grids.

Im Sinne eines Geschäftsmodells werden innerhalb einer VO Ressourcen angeboten und verbraucht, die lokal nicht, beziehungsweise nicht ausreichend vorhanden sind. VOs unterstützen die Entwicklung von ausgangs großen, statischen Unternehmen hin zu flexiblen, temporären Netzwerken von Einzelpersonen, [108].

VOs können sich über mehrere Standorte, beziehungsweise administrative Domains erstrecken, siehe Abbildung 1.5. Hierbei können heterogene Kommunikations- und Sicherheitsprotokolle, sowie verschiedene Isolationsmechansimen (Virtuelle Maschinen siehe Kapitel 1.3.2, leichtgewichtige Virtualisierung siehe Kapitel 4.3.2) unter anderem zum Einsatz kommen. Interoperabilität heterogener Mechanismen ist daher die zentrale Herausforde-

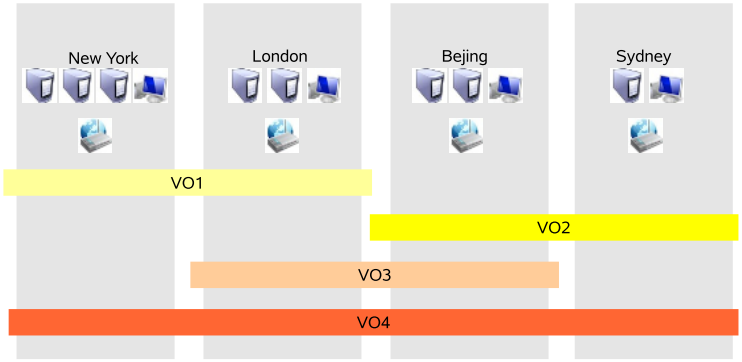

Abbildung 1.5: Virtuelle Organisationen

rung bei der Realisierung des VO-Konzeptes. In diesem Kontext müssen insbesondere VO-Anwendungs-Entwickler unterstützt werden, die stets auf gleichbleibende Funktionalitäten wie Sicherheit, Ressourcenverwaltung und Fehlererkennung angewiesen sind, um Quality of Service auch auf Ressourcen mit heterogenen und oft sehr dynamischen Charakteristiken umsetzen zu können.

Ferner wird das Grid als erweiterbare Menge von Griddiensten betrachtet. Dabei können Griddienste zu höherwertigen Diensten aggregiert werden, um Anforderungen einer VO zu erfüllen. Höherwertige aggregierte OGSA-Dienste bilden beispielsweise verteiltes Datenmanagement, Workflowdienste, Kontroll-, Überwachungs- und Sicherheitsdienste. Im Folgenden werden elementare Dienste eines Grids erläutert.

1.4.3 Ressourcen-Management

Ressourcen-Management ist der Schlüssel dafür, Aufgaben im Grid effizient und automatisiert zu verteilen [83]. Einerseits muss die Adressierbarkeit von Ressourcen gewährleistet werden, andererseits müssen die Nutzungsgrenzen von Ressourcen überwacht werden. Beide Aspekte setzen die Erfassung aller Ressourcen voraus, die beispielsweise anhand eines verteilten Ressourcenmonitors, wie beispielsweise Ganglia [98], ermöglicht werden kann.

Hinsichtlich der Adressierung von Ressourcen existieren unterschiedliche Ansätze zu Suchverfahren und involvierten Komponenten. In der Literatur wird überwiegend ein *dezentralisierter* Ansatz vorgeschlagen, der aus verteilten Verzeichnissen besteht. Während UDDI [33] einen Defacto Industrie-Standard zur Erkundung von Web-Services darstellt, haben Anforderungen nach strenger Verzeichnis-Replikation und das Fehlen autonomer Kontrolle dessen großflächigen Einsatz behindert. Daher wird in [19] ein *Rendezvous-Mechanismus* zwischen verschiedenen UDDI Verzeichnissen vorgeschlagen, der auf verteilten Hash-Tabellen basiert. Dies *erlaubt dem Benutzer* mehrfach Anfragen zu stellen, während gleichzeitig Organisationen autonome Kontrolle über die Einträge besitzen.

Um die Komplexität des Grids vor dem Benutzer zu verbergen, ist der Aspekt autonomer,

nicht durch Benutzer initiierter, Suche und Auswahl von Ressourcen von großer Bedeutung. In einem agentenbasierten Ansatz wird in drei Agententypen, Dienstanbieter, Dienstanfrager und zwischengelagerter Agent, unterschieden, [140].

Ein bedeutender Aspekt ist zudem, Agenten entprechend auf Knoten zu platzieren. In [103] agiert *ein* Agent auf einem Grid-Knoten, beispielsweise einem Cluster, nach zentralistischem Muster, oder es existieren mehrere Agenten, die Grid-Knoten übergreifend miteinander interagieren, [103]. Sind mehrere Agenten vorhanden, wird in [111] der geeignetste Knoten zwischen den Agenten untereinander ermittelt.

MapReduce [36] ist ein Programmiermodell, welches Programme parallelisiert, gleichzeitig Details der Parallelisierung, Fehlertoleranz, Lokalitäts-Optimierung und der Lastenverteilung (engl. Load-Balancing) vor dem Programmierer verbirgt. Es eignet sich zur Lokalisierung von Ressourcen im Grid, die bestimmten Sucheigenschaften entsprechen.

1.4.4 Job-Management

Ein Job-Management-System (JMS) schafft eine Umgebung zur Ausführung und Kontrolle eines Jobs im Grid. Der Begriff Job Management System wird in der Literatur auch mit Resource Management System, Workload Manager, et cetera gleichgesetzt, [117]. Hieraus wird die enge Kooperation von Job- und Ressourcen-Management ersichtlich.

Ein JMS agiert knotenübergreifend, um Jobs[12] auf lokalen und entfernten Ressourcen auszuführen. Aufbauend auf den Informationen eines Ressourcen-Management-Systems, platziert ein JMS Jobs in Warteschlangen und startet sie in Abhängigkeit von der Auslastung der zur Verfügung stehenden VO-Knoten. Eine Balance zwischen verfügbaren Ressourcen und ausgeführten Anwendungen erhöht die zuverlässige Anwendungsausführung, da keine Anwendung infolge von Ressourcenknappheit beeinflusst wird. Gleichzeitig optimiert es die Knotenauslastung.

Weiterhin bietet ein JMS eine Schnittstelle um einen Job zu starten, zu kontrollieren und zu beenden (Job-Signale) und auf Jobdaten auf den VO-Knoten zuzugreifen. Ein JMS leitet wichtige Zustandsänderungen, wie Fehlermeldungen oder Warnungen, über die Schnittstelle an den Benutzer weiter.

Im Middleware-Bereich existieren mehrere JMS-Umsetzungen. Condor, [143], ist ein spezialisiertes Workload-Management-System vorzugsweise für HTC (High Tech Computer) Anwendungen. Agenten und Ressourcen bieten ihre Charakteristiken sogenannten Matchmakern an, sodass Ressourcen zwischen Anbieter und Klient vermittelt (engl. to broker) werden können. Condor erlaubt bei Bedarf verschiedene Cluster miteinander zu verbinden (Demand Computing). Jobs werden bei Condor in Warteschlangen verwaltet und können gesichert, wiederhergestellt und migriert werden.

[12] Ein Job bezeichnet eine im Grid ausgeführte, sequentielle oder verteilte, Anwendung. Sie besteht pro involviertem Grid-Knoten aus einem oder mehreren Prozessen.

Globus Resource Allocation and Management (GRAM),[50], ist die wichtigste Komponente von Globus. Benutzer können entfernte Jobs lokalisieren, einreichen, überwachen und beenden. GRAM erlaubt entferntes Ausführungsmanagement mit besonderem Fokus auf zuverlässiger Ausführung, Zustandsüberwachung, Zertifikats-Management und Dateiverwaltung, basierend auf RFT und GridFTP [121]. GRAM ist kein Job Scheduler, bietet jedoch ein Protokoll, um mit verschiedenen Batch/Cluster Job Schedulern zu kommunizieren.

Sun Grid Engine (SGE), [60], ist ein JMS, welches freie Prozessorressourcen verwaltet. Es bietet Unterstützung für die automatische Cluster-Erweiterung um Ressourcen, die nur in bestimmten Zeiträumen verfügbar sind, beispielsweise nachts. Jobs können anhand komplexer Beschreibungen detailliert definiert werden, unter Einbezug von harten und weichen Kriterien. SGE erlaubt Ressourcen für Jobs zu reservieren, preemptives Job Scheduling durchzuführen und verwendet die Distributed Resource Management Application API (DRMAA) als OGF Standard zur Einreichung und Kontrolle von Jobs.

1.4.5 Daten-Management

Daten-Management bezieht sich auf Inhalte in flüchtigem und/oder persistentem Speicher. Daten sicher, zuverlässig und effizient zu transferieren gehört zu den fundamentalen Anforderungen datenintensiver Anwendungen [8]. Da lokale Datenkopien *Transferzeiten* im Grid reduzieren können, muss auch das Registrieren, Lokalisieren und Verwalten *mehrerer Kopien eines Datensatzes* hinzugezählt werden. Soll gleichzeitig *Daten-Konsistenz* gewährleistet werden, kann ein erheblicher Programmieraufwand entstehen. Es herrscht Einigkeit darüber, dass ein Programmiermodell höherer Ebene vonnöten ist, wonach die beiden erwähnten Herausforderungen nicht vom Anwendungs-Programmierer, sondern von Gridwerkzeugen und dem Laufzeitsystem gelöst werden sollen, [7]. Letzteres erleichtert die Entwicklung von Grid-Anwendungen.

Distributed Shared Memory (DSM) Systeme, wie Kerrighed [107], JuxMem [17] und Object Sharing Service (OSS) [109] erlauben mehreren Benutzern nebenläufig und knotenübergreifend auf gemeinsame Hauptspeicher-Inhalte variierender Größe (Speicherseiten-Objekt, eine oder mehrerere Speicherseiten), zuzugreifen, ohne Anwendungsprogrammierer mit der Implementierung eines Konsistenzmodells zu belasten. In [106] werden Checkpoints im Hauptspeicher anderer Knoten gespeichert, die Checkpointkopieverwaltung wird dabei in das Konsistenzprotokoll einer Cache-Only-Memory-Architektur (COMA)[13] integriert.

Existierende Grid-Datenmanagementsysteme, wie DataGrid [134], bieten Benutzern Zugriff auf Daten an, die sich auf entfernten Speicherknoten und Dateisystemen befinden. Diese Systeme bestehen meist aus einer Client-Server-Architektur und sind aus Speicherressourcen-, Replikate- und Metadatenkatalog-Komponenten zusammengesetzt. In Data-

[13] Bei COMA wird der Rechnerhauptspeicher in einen großen Adressraumcache konvertiert, der durch mehrere Rechner genutzt werden kann.

Grid werden Werkzeuge zum Replizieren von Dateien realisiert. Hierbei müssen jedoch Dateien *vollständig* von einer Speicherressource in das lokale Dateisystem herunterladen werden, bevor darauf zugegriffen werden kann. Weiterhin wird Replikakonsistenz nur unzureichend gewahrt, sodass lediglich das write-once Schreibzugriffsmuster möglich ist. Demnach dürfen dem einzigen Schreibvorgang nur noch Leseoperationen folgen, was eine starke Beschränkung darstellt. Zuzüglich stellt bei DataGrid ein zentrales Datenzentrum ein Single Point of Failure (SPF) dar.

Alternativ hierzu realisiert XtreemFS ein objektbasiertes Dateisystem [76], [77]. XtreemFS verwaltet sogenannte Dateisystemvolumes, einhängbare Dateisysteme, die bei einem gridweiten Verzeichnisdienst registriert werden, deren *Inhalte repliziert und Blöcke einer Datei auf mehrere Knoten verteilt* werden (Striping). Durch parallelen Zugriff auf alle beteiligten Knoten entstehen effizientere Ein-/Ausgabezeiten, im Gegensatz zur Verwendung eines Knotens. Metadaten und Blockinhalte werden voneinander getrennt und von redundanten Komponenten verwaltet, welche über einen verteilten Verzeichnisdienst referenziert werden, sodass ein SPF bei Absturz einer Komponete ausgeschlossen wird. Interne Koordinierung erlaubt POSIX Semantiken beizubehalten, insbesondere im Falle nebenläufigen Zugriffs auf Repliken. Anwendungen müssen demnach für die Ausführung im Grid nicht abgeändert werden. XtreemFS gewährt zusätzlich Sicherheit, indem Volumezugriffe ausschließlich gültigen VO-Nutzern vorbehalten ist.

1.4.6 Sicherheit

Grid-Sicherheit berührt Aspekte wie Identitäts-, VO-Mitglieds-, Ressourcen- und Datenspeicherungs-Management, Kommunikation und Isolation [88]. Aus Komplexitätsgründen kann im Rahmen dieser Arbeit nur ein kleiner Ausschnitt näher beleuchtet werden.

Sicherheit im Grid ist eng mit VO-Sicherheit verknüpft, da VOs Zellen des Grids sind. Das heißt, Mitgliedern einer VO A muss die Nutzung gemeinsamer Ressourcen, beispielsweise mit Mitgliedern einer VO B, vollständig verborgen bleiben. VO-Sicherheit erstreckt sich über Grid- und Knoten-Ebene und kann u.a. durch Integration allgemein existierender und Linux-spezifischer Sicherheitsmechansimen, wie Pluggable Authentication Module (PAM), Linux Capabilities, Linux Security Module Framework (LSM) oder Access Control Lists (ACL), realisiert werden. Im Folgenden werden vier elementare Aufgaben einer Grid-Sicherheits-Architektur erläutert.

1.4.6.1 Vertrauensaufbau

Im Zentrum sicheren VO-Managements steht das Aufbauen von Vertrauen (engl. trust) zwischen Benutzer und Maschine [149]. Hierzu müssen sich beide Seiten *gegenseitig authentisieren*, damit später nur gültige VO-Mitglieder auf vertrauenswürdige VO-Ressourcen zugreifen können. Eine Grid-Sicherheitsinfrastruktur muss dabei auf Integration heteroge-

ner Authentisierungsmechanismen, wie Kerberos, Public Key Infrastructure (PKI) oder Passwort-basierter Techniken, achten, da bestehende Sicherheits-Infrastrukturen von Unternehmen, die einer VO beitreten wollen, nicht modifiziert werden sollen.

In einer PKI-Umgebung kann Vertrauen anhand *hierarchischer Vertrauensmodelle mit und ohne Gegenzertifizierung einzelner Komponenten* hergestellt werden, [93]. Diese Vertrauensmodelle leiden jedoch unter Problemen der Konfiguration und Verwendbarkeit, da Zertifikate und öffentliche Schlüssel von Wurzelkomponenten korrekt platziert werden müssen. Spezielle Techniken werden benötigt, die eine *automatisierte* Verteilung von Zertifikaten und Schlüsseln ermöglichen, um Gridkomplexität vor den Administratoren und Benutzern zu verbergen. Zu letzterem zählt Single Sign On (SSO), wonach sich ein Benutzer nur *einmal* registriert, relevante Sicherheitsinformationen jedoch automatisch auf *allen* VO-Ressourcen installiert werden.

1.4.6.2 Autorisierung und Authentisierung

Im Anschluss an eine erfolgreiche Authentifizierung, wird ein VO-Mitglied autorisiert. Dies beinhaltet, darüber zu entscheiden, welche Benutzeraktionen ausgeführt werden dürfen und wird in Abhängigkeit vorliegender Identitäten, Attribute und Zeugnisse entschieden. Da Gridsemantiken wie VO-Benutzer-IDs nicht auf Linux-Systemen existieren, müssen Abbildungen auf lokale Benutzer pro Grid-Knoten vorgenommen werden. Dies kann beispielsweise mithilfe eines gridspezifischen PAM-Moduls realisiert werden, siehe [105], zweites Kapitel.

1.4.6.3 Kommunikation

Sichere Kommunikation muss für die Kommunikation zwischen Grid-Entitäten (Dienste, VO-Benutzer, Ressourcen) auf Systemebene explizit abgesichert werden, da der Datenverkehr über unsichere Kanäle wie dem Internet verläuft. Gridanwendungen hingegen sind selbst für die Absicherung der Kommunikation verantwortlich.

Die Einhaltung von Datenvertraulichkeit und -integrität ist bei Kommunikationssicherheit wichtig, darf jedoch nicht mit hohen Latenzen und hohem Managementaufwand verbunden sein.

Basierend auf den Informationen und Algorithmen, die allen Grid-Entitäten zur Verfügung stehen, wird eine Technik zur abgesicherten Kommunikation bestimmt. Insofern beglaubigte öffentliche Zertifikate, wie öffentliche Schlüssel, zugänglich sind und Implementierungen der Algorithmen wie DES, AES oder RSA vorliegen, kann beispielsweise Transport Layer Security (TLS) verwendet werden, um kommunizierende Entitäten wechselseitig zu authentisieren. Wichtig ist jedoch, dass mithilfe von TLS Daten ver- und entschlüsselt werden können und somit Datenvertraulichkeit und -integrität erzielt wird.

1.4.6.4 Ressourcen-Isolation

Stürzt die Ausführungsumgebung von VO-Benutzer A ab, darf es nicht zum Absturz jener von VO-Benutzer B kommen (Ausfallssicherheit). Ein Benutzer soll zudem auf einem gemeinsamen Gridknoten Beobachtungen durchführen und Ausgabedaten produzieren können, die nicht durch Interaktionen anderer Benutzer auf demselben Knoten beeinträchtigt werden (Unterbindung von Interferenzen).

Die Isolation benutzergebundener Umgebungen kann beispielsweise mit Virtualisierung von Ressourcen erzielt werden. Derzeit existierende Technologien umfassen Betriebssystem-Container, siehe Kapitel 4.3.2, als auch unterschiedliche Formen virtueller Maschinen, beispielsweise Hardware-Emulation und Para-Virtualisierung, siehe Kapitel 1.3.2.

1.4.7 Referenzarchitektur

In der Referenzarchitektur, welche das Fundament der Grid-Checkpointing-Architektur in Kapitel 3.1 darstellt, besteht ein Job aus einer oder mehrerer sogenannter *Job-Einheiten*, die jeweils auf separaten Gridknoten, das heißt einem einzelnen PC oder einem SSI-Cluster, ausgeführt werden. Jobs werden auf Gridebene vom sogenannten Job-Manager verwaltet, der ein verteilter Dienst ist und auf jedem Gridknoten ausgeführt wird. Ein Job wird jedoch genau von einer Job Manager-Instanz kontrolliert, welche die Adressen aller Einheiten eines Jobs kennt, Signale an sie sendet und Zustandsänderungen von ihnen empfängt und verarbeitet. Im Kontext der Jobverwaltung interagiert der Job-Manager mit dem Ressourcenmanagementsystem (RMS), um Job-Einheiten auf verfügbare und kompatible Gridknoten innerhalb einer VO zu platzieren. Jobmetadaten werden in einem Verzeichnisdienst zum gridweiten Zugriff abgelegt.

Auf Gridknotenebene wird eine Job-Einheit vom sogenannten Job-Einheit-Manager verwaltet, der auf jedem Gridknoten ausgeführt wird, jedoch kein verteilter Dienst ist. Da eine Job-Einheit aus einem nativen Linux-Prozess beziehungsweise aus einer Linux-Prozessgruppe, siehe Kapitel 4.2 besteht, besitzt der Job-Einheit-Manager (JEM) die zur Verwaltung notwendige Prozesssicht. Prozesse stellen die, in dieser Architektur verwendete, Ausführungsumgebung [54] auf Gridknotenebene dar.

In der Referenzarchitektur wird ein verteiltes Dateisystem verwendet, um auf persistente Daten ortstransparent zugreifen zu können. Dieses Dateisystem gewährleistet Datenreplikation, um einerseits die Wahrscheinlichkeit von Datenverlusten, infolge von Knotenausfällen, zu maskieren und andererseits Datenstriping, um Datenzugriffszeiten zu optimieren.

Nur authentifizierte und autorisierte Benutzer dürfen Jobs einreichen und auf Ressourcen einer VO ausführen. Job- und Job-Einheit-Manager unterstützen die dafür benötigten Sicherheitsmaßnahmen. Damit bei einem lokalen oder entfernten Restart die vor Checkpointing gültige Benutzerkennung verwendet und damit insbesondere die Überprüfung der Zugriffsrechte auf Prozesse, Dateien, et cetera durch einen Checkpointer erfolgreich durch-

geführt werden kann, wird die Benutzerkennung der Gridebene (JobID) auf eine Benutzerkennung der Gridknotenebene (uid, gid) mithilfe eines PAM-Moduls *bijektiv* abgebildet. Um mehr Gridbenutzer zu ermöglichen als bisher mit einer lokalen darstellbar sind, müssen uids mit einer Länge von 64 Bit verwendet werden.

1.5 Zielsetzungen dieser Arbeit

Diese Arbeit beschreibt die Konzeption und Implementierung von Grid-Fehlertoleranz. Grid-inhärente Heterogenität bei Hard- und Softwarekomponenten und eine hohe Ressourcendynamik beeinflussen die Architektur maßgeblich. Die Beiträge dieser Arbeit sind:

1. Die Konzeption und Implementierung einer Grid-Checkpointing-Architektur (GCA) unter Einbezug existierender, heterogener Checkpointer.
2. Ein Mechanismus, um Zustände eines Kommunikationskanals an beiden Kanalenden mit heterogenen Checkpointern zu sichern und wiederherzustellen.
3. Die Realisierung adaptiven Checkpointings durch den abwechselnden Einsatz unterschiedlicher Checkpointingstrategien.

1.6 Aufbau der Arbeit

Die Dissertation gliedert sich in neun Kapitel. Das erste Kapitel erläuterte bereits wesentliche theoretische Grundlagen der Fehlertoleranz und beschrieb existierende Techniken, um Fehlertoleranz zu realisieren. Darüber hinaus wurden wichtige Bestandteile einer Grid Computing-Umgebung vorgestellt, die, als Basis für eine Grid-Checkpointing-Architektur im späteren Verlauf der Arbeit, dienen.

In Kapitel zwei werden Eigenschaften und darauf aufbauend Realisierungsformen von Grid Checkpointing untersucht.

Das dritte Kapitel bildet den Schwerpunkt der Arbeit. Darin wird eine Grid-Checkpointing-Architektur (GCA) beschrieben, in deren Zentrum die Einbindung heterogener Checkpointerpakete steht.

Konsistente Anwendungszustände können nur erzeugt werden, wenn Semantiken der Grid- und Knotenebene korrekt aufeinander abgebildet werden. In Kapitel vier werden die Beziehungen existierender Prozessgruppen und Checkpointer, in Verbindung mit relevanten Techniken der Virtualisierung, dargestellt.

In Kapitel fünf wird diskutiert, welche Bedeutung Callbacks im Grid-Checkpointing-Kontext besitzen und wie sie integriert werden können.

Kapitel sechs stellt einen Mechanismus vor, um Zustände eines Kommunikationskanals mit heterogenen Checkpointern pro Kanalende zu sichern und wiederherzustellen, ohne Anwendungen und Checkpointer zu modifizieren.

Kapitel sieben beschreibt zwei Varianten, mithilfe derer adaptives Checkpointing realisiert werden kann. Hierbei steht der beidseitige Wechsel zweier Checkpointingprotokolle und inkrementelles Grid-Checkpointing im Vordergrund.

In Kapitel acht werden relevante Performanz-Messwerte der GCA vorgestellt und diskutiert.

Zum Abschluss werden die Ergebnisse dieser Arbeit in Kapitel neun zusammengefasst.

2 Grid-Checkpointing

In diesem Kapitel werden konzeptuelle Aspekte des Grid-Checkpointings beschrieben. Hierzu gehören Anwendungsfälle sowie Anforderungen verschiedener Einflussfaktoren. Elementare Anforderungen werden in Bezug zueinander gesetzt und danach untersucht, inwieweit sie miteinander vereinbar sind. Daraus lassen sich unterschiedliche Ansätze zur Realisierung von Grid Checkpointing ableiten.

2.1 Anwendungsfälle von Grid-Checkpointing

Grid-Checkpointing entspricht dem Sichern und Wiederherstellen von Grid-Anwendungs-Zuständen. Die Grid-Checkpointing konstituierenden Teilsequenzen *stop, checkpoint, resume und restart* können dazu verwendet werden, um *Job-Suspendierung*, *Job-Migration* und *Job-Fehlertoleranz* zu realisieren.

Job-Suspendierung ermöglicht einen, über mehrere Gridknoten verteilten, Job auf einfache Weise, beispielsweise mit *einem Benutzerkommando*, anzuhalten. Optional kann der Job-Zustand hierbei gesichert werden. Zu einem vom Benutzer oder vom System bestimmten Zeitpunkt kann die Job-Berechnung fortgesetzt werden.

Job-Suspendierung erlaubt, Reparaturen oder Erweiterungen der Rechner-Hardware vorzunehmen, ohne die Datenkonsistenz des Jobs zu beeinträchtigen. Ein Job kann beispielsweise daran gehindert werden, Ergebnisdaten zu generieren, solange bis genügend Festplattenplatz eingebaut wurde. Weiterhin kann eine *lokale Job-Ablaufplanung (engl. Scheduling)*[1] ermöglicht werden, indem Jobs niedrigerer Priorität zugunsten Jobs höherer Priorität schlafen gelegt und zu einem späteren Zeitpunkt weiter ausgeführt werden.

Bei *Job-Migration* wird der Zustand eines Jobs, der sich über einem beziehungsweise mehreren *Quellgridknoten* befindet, konserviert, die Ausführung unterbrochen[2] und der Zustand *unmittelbar* auf einem beziehungsweise mehreren *Zielgridknoten* wiederhergestellt. Job-Sicherung und Wiederherstellung werden, zeitlich gesehen, *miteinander gekoppelt*. Die extrahierten Zustände werden mithilfe verteilter Dateisysteme oder P2P-Dienste auf den Zielgridknoten verfügbar gemacht. Unmittelbar nach der Job-Migration werden die zur Speicherung der extrahierten Zustandsdaten verwendeten Ressourcen freigegeben.

[1] Hierbei findet keine Migration statt.
[2] Bei Live-Migration erfolgt keine zwischenzeitliche Anwendungs-Unterbrechung.

Ein typischer Anwendungsfall von Job-Migration ist die Job-Ablaufplanung, um Lastenverteilung (engl. load balancing) zu realisieren. Ein sogenannter Scheduler entscheidet darüber, wann ein Job verlagert werden soll. Differenzen der Ressourcenauslastung oder variierende Ressourcen-Nutzungsgebühren (nachts sind die Ressourcen kostengünstiger) können hierfür ausschlaggebend sein.

Bei *Job-Fehlertoleranz* werden Job-Zustände in periodischen Abständen gesichert, um deren Verlust, infolge von Knotenausfällen, zu verhindern.

In der Realität können vor allem über mehrere Knoten verteilte und/oder Langzeitanwendungen vor dem Rückfall auf den initialen Zustand bewahrt werden. Die Job-Sicherung und -Wiederherstellung kann eingesetzt werden, um Anwendungsfehler effizienter zu erkennen und zu beheben (Debugging). In diesem Zusammenhang werden *gezielt* Jobzustände der Vergangenheit erzeugt, bei denen in absehbarer Zeit Fehler auftreten [141]. Damit lassen sich insbesondere solche Job-Zustände auf einfache Art und Weise reproduzieren, die bei Anwendungen mit integrierten Nichtdeterminismen entstehen, beispielsweise transaktionale Speicher- oder Multithreadinganwendungen, und nur sehr zeitaufwendig reproduzierbar sind.

2.2 Anforderungen an Grid-Checkpointing

Eine Grid-Checkpointing-Architektur muss existierende Fehlertoleranztechniken, Grid- und Anwendungseigenschaften, aber auch Benutzer- und Programmiererpreferenzen bei der Konzeption berücksichtigen.

2.2.1 Konsistenz

Konsistente Abbilder, siehe Kapitel 1.1.3, sind obligatorisch, um nur die Zustände im Fehlerfall wiederherzustellen, die in der fehlerfreien Aufführung auftreten könnten. Weiterhin belastet der, nicht mit einem konsistent Abbild verbundene, Fehlertoleranzaufwand die Systemausführung beispielsweise in Form von Festplattenein-/ausgabe und Netzwerkverkehr und die Anwendungsausführung beispielswesie durch Synchronisierung zusätzlich.

2.2.2 Anwendungsunterstützung

Grid-Checkpointing sollte möglichst viele *Anwendungstypen unterstützen.* Interaktive, graphische, rechen-, daten- oder kommunikationsintensive Anwendungen stellen jedoch unterschiedliche Anforderungen an einen Sicherungs- und Wiederherstellungsmechanismus:

- interaktiv: Eingehende Nachrichten, die ein System von der Außenwelt erhält, müssen aufgezeichnet werden, damit sie bei der Wiederherstellung rekonstruiert werden kön-

nen, insbesondere dann, wenn die Außenwelt sie nicht rekonstruieren kann. Hierzu bieten sich Log-basierte Checkpointingprotokolle an.

- graphisch: Zustände des X-Servers müssen in die Sicherung und Wiederherstellung einbezogen werden. Im Gegensatz zu existierenden Anwendungscheckpointern bieten ausschließlich Systemcheckpointer hierfür Unterstützung.
- rechenintensiv: Um den Berechnungsfortschritt durch Checkpointing nicht zu stark zu beeinträchtigen, bietet sich nebenläufiges Sichern (COW Checkpointing) an.
- datenintensiv: Dateien oder das gesamte Dateisystem müssen vom eingesetzten Checkpointer gesichert und wiederhergestellt werden können. *Optionales Sichern großer Dateien oder inkrementelles Sichern großer Hauptspeicherinhalte* kann zu effizienterem Checkpointing führen.
- kommunikationsintensiv: In Abhängigkeit des Checkpointingprotokolls müsssen bei vielen Nachrichten viele Determinanten oder Abhängigkeitsinformationen aufgezeichnet werden, was die Nachrichtenauslieferung während der fehlerfreien Ausführung beeinträchtigen kann. Bei MPI-Anwendungen sollten vorzugsweise MPI-Checkpointer verwendet werden, die MPI-Semantiken interpretieren können.

2.2.3 Heterogenität

Im Grid-Comuting werden heterogene Komponenten bezüglich der installierten Software, der Knoten-Hardware und Netzwerkarchitektur miteinander verknüpft. Sollen gridweit gleichwertige, höhere Dienste, auf Basis unterschiedlicher Komponenten, realisiert werden, muss *Heterogenität* als Grid-inhärente Eigenschaft verstanden und von einer Grid-Checkpointing-Architektur berücksichtigt werden. Im Checkpointing-Bereich existieren unterschiedliche Checkpointingprotokolle, wobei sich keines von ihnen optimal in allen Anwendungsfällen verhält. Deshalb müssen verschiedene Protokolle unterstützt werden. Zudem existieren in der Praxis mehrere Checkpointerimplementierungen mit unterschiedlicher Funktionalität, oft an spezifische Hardware- und Software Konstellationen gebunden, siehe Unterkapitel 1.3.

2.2.4 Skalierbarkeit

Der Begriff Skalierbarkeit ist im Checkpointingbereich mehrdimensional.

Definition 10: Skalierbarkeit der Anwendungssicherung und -wiederherstellung. Die Anwendungssicherung und -wiederherstellung skaliert dann, wenn der Aufwand relevanter Fehlertoleranzoperationen höchstens linear mit dem Umfang der zu sichernden und wiederherzustellenden Ressourcen und der Verteilung der Anwendung auf mehrere Gridknoten wächst.

Je größer die Prozessadressräume sind, desto mehr Ein-/Ausgabe findet statt, wenn Abbilder geschrieben und eingelesen werden. Je mehr verteilte Prozesse existieren, desto mehr Kontrollnachrichten müssen versendet werden. Je weiter die Prozesse voneinander entfernt sind, desto größer können die Auswirkungen von Netzlatenzen und variierenden Bandbreiten sein. Das Ein-/Ausgabeverhalten, Netzwerklatenzen und -bandbreiten sind jedoch Einflussfaktoren, die im Grid, aufgrund dynamischer Ressourcenverfügbarkeit, *nicht konstant* bleiben und unvorhersehbar sind. Unter diesen Umständen gibt es keine ultimative Checkpointingstrategie. Die Skalierbarkeit der Anwendungssicherung und -wiederherstellung ist damit eng an adaptives Checkpointing, siehe Kapitel 7, geknüpft.

Definition 11: Skalierbarkeit des Gridknotencheckpoints. Der Checkpoint, beziehungsweise die Abbilddatei skaliert, wenn sie auf unterschiedlichen Gridknoten interpretiert werden kann und damit zu einem erfolgreichen Restart führt.

In diesem Zusammenhang spielt die *Interpretierbarkeit eines Checkpoints* eine große Rolle. Da jeder Checkpointer Zustände verschiedenartig abspeichert, kann eine Abbilddatei nur von seinem Erzeuger interpretiert werden, sodass Anwendungszustände korrekt wiederhergestellt werden. Weil nicht jeder Checkpointer auf jedem Gridknoten installiert ist, sind Abbilddateien, ohne weitere Maßnahmen, nur auf einer Untermenge aller Gridknoten verwendbar.

2.2.5 Transparenz

Anwendungstransparenz bedeutet, dass der Anwendungsquelltext nicht abgeändert und neukompiliert werden muss, um gesichert und wiederhergestellt werden zu können. Dies ist vor allem für Legacy-Anwendungen entscheidend, da integrierte Fehlertoleranz mit hohem Implementierungsaufwand, beziehungsweise einer Ausführungsunterbrechung, verbunden sein kann, was vermieden werden muss.

Wird eine Anwendung nicht abgeändert, besteht jedoch die Gefahr, mehr Ressourcen zu sichern, als tatsächlich notwendig ist[3]. Dies tritt ein, weil ein anwendungsexterner Checkpointer kein anwendungsinternes Wissen um sicherungsrelevante und -irrelevante Ressourcen besitzt. Hingegen können Anwendungsprogrammierer anwendungsinternes Wissen mit Hilfe der in Kapitel 5 vorgestellten Callbacks integrieren, um Ressourcen *gezielt* zu sichern.

2.2.6 Sicherheit

Sicherheit stellt im Checkpointingkontext eine große Herausforderung dar. Gesicherte Zustände dürfen nur jenen Benutzern zugänglich sein, die vor der Sicherung Rechte für diese Zustände besaßen. Eine Überprüfung auf authentische und autorisierte Benutzer ist daher

[3] Eine Datenbankanwendung mit Datensätzen im Terabytebereich sollte aus Effizienzgründen nicht in kurzen Intervallen gesichert werden.

unabdingbar.

Nach einem Checkpoint können zwischenzeitliche Änderungen der Systemumgebung auftreten, welche von einem *statischen Checkpoint* nicht reflektiert werden können. Datenintegrität kann gezielt manipuliert werden. Jedoch kann auch der legetimierte Austausch von Bibliotheksversionen, beispielsweise die Aktualisierung durch einen Administrator, zu Restartkonflikten führen, insofern vor einer Sicherung eine ältere Version verwendet, die jedoch nicht abgespeichert wurde und bei Restart nicht mehr verfügbar ist. Demnach kann es auch bei Migrationen zu Konflikten kommen, wenn zuvor kein Abgleich auf kompatible Bibliotheksversionen vorgenommen wurde.

Weiterhin sind im Checkpoint gesicherte Dateirechte, die mit aktuellen Rechten im Dateisystem nicht mehr übereinstimmen, problematisch, da der Dateizugriff nach einem Restart verhindert werden kann.

2.2.7 Effizienz und Performanz

Um Speicherressourcen, Netzwerkbandbreiten und andere Systemressourcen effizient zu nutzen[4], gleichzeitig den Checkpointingverhead auf System- und Anwendungsausführung zu minimieren, muss die optimale Checkpointingstrategie ausgewählt werden. Daher sind Effizienz und Performanz eng mit *adaptivem Checkpointing* verbunden.

2.2.8 Zuverlässigkeit von Grid-Checkpointing

Der Ausfall einer zentralen Grid-Checkpointing-Komponente stellt einen Single Point of Failure dar. Er kann im Fehlerfall zur System-, beziehungsweise Anwendungsblockierung führen und muss somit durch geeignete Techniken, wie beispielsweise Replikation, siehe Kapitel 1.2.5, verhindert werden.

2.2.9 Bedienbarkeit

Die Komplexität von Grid-Checkpointing muss vor einem Benutzer in stärkerem Maße, als vor einem Administrator verborgen werden. Die Gefahr eines Fehlverhaltens muss, wenn möglich, verhindert werden. Werden beispielsweise Abbilddateien gelöscht oder manipuliert, sind Systembeeinträchtigungen zu erwarten, da Checkpointing überwiegend auf Systemebene ausgeführt wird[5]. Wird die Prozesswiederherstellung vorzeitig beendet, bei-

[4] Eine Garbage Collection ist beispielsweise für unkoordiniertes Checkpointing, um nicht zu einem globalen Checkpoint gehörende Abbilder entfernen zu können.

[5] Liegen unvollständige oder unbrauchbare Daten vor, kann es zur Systemblockade oder -absturz kommen, wenn das System dies nicht erkennt und entsprechend reagiert. An dieser Stelle muss Fehlertoleranz selbst auf Fehlertoleranzmechanismen angewandt werden, siehe Kapitel 3.6.

spielsweise durch bewusstes oder unbewusstes Senden entsprechender Signale, entstehen ähnliche Konsequenzen. Einfache Bedienbarkeit berührt daher Sicherheitsaspekte und den Automatisierungsgrad von Grid-Checkpointing.

2.3 Vereinbarkeit der Grid-Checkpointing-Anforderungen

Eine Grid-Checkpointing-Architektur, welche alle der in Kapitel 2.2 diskutierten Kriterien realisiert, ist schwierig, da einige in der Praxis schwer miteinander vereinbar sind. Konkurierende Ziele werden nachfolgend diskutiert.

2.3.1 Heterogenität, Skalierbarkeit und Abbildformate

Es existiert kein ultimativer Checkpointer, der alle Ressourcen sichern und wiederherstellen kann.

Zusätzlich kann nicht davon ausgegangen werden, dass *alle Grid-Knoten mit einem einheitlichen Checkpointer ausgerüstet* sind. Dies ist in der Praxis die Regel, da einerseits bestimmte Checkpointer an bestimmte Gridknoten gebunden sind, beispielsweise BLCR an einen PC, der LinuxSSI Checkpointer an ein SSI Cluster, und ein Grid heterogene Gridknoten zusammenfasst. Andererseits liegt auch kein einheitlicher Checkpointer vor, wenn auf zwei Gridknoten jeweils eine *unterschiedliche Version des gleichen Checkpointertyps* installiert ist[6]. Wenn Checkpointer A ein Abbild anlegt, verwendet er ein Datenformat, welches Checkpointer B nicht, beziehungsweise nicht vollständig, interpretieren kann.

Um im Kontext verschiedener Abbildformate Skalierbarkeit von Gridknotencheckpoints, siehe Kapitel 2.2.4, zu erreichen, kann aus theoretischer Sicht das Abbild eines Checkpointers A in das Abbild für Checkpointer B *übersetzt* werden. Abbildübersetzung ist jedoch nicht in jedem Fall möglich, beispielsweise wenn alle Zielformatdaten nicht aus den Quellformatdaten generiert werden können[7]. Andererseits verursacht eine Abbildübersetzung Daten- und Berechnungsaufwand und beeinträchtigt dadurch die Gesamtperformanz.

2.3.2 Heterogenität, Portabilität und Plattformen

Eine auf Heterogenität ausgerichtete Grid-Checkpointing-Architektur integriert auch Knoten, die virtuelle Maschinen als Checkpointer einsetzen, siehe [80]. Bei SUNs Virtualbox wird jedoch die Migration eines VM-Abbilds von einem Intel- zu einem AMD-System nicht

[6] Die kleinste Änderung der Abspeichersequenz von Checkpointdaten genügt, um die korrekte Dateninterpretation außer Kraft zu setzen.

[7] Informationen zu LinuxSSI-spezifischen KDDM-Strukturen, [100], können nicht aus einem BLCR Abbild abgeleitet werden, weil sie nur in LinuxSSI vorhanden sind.

erlaubt [145], sodass eine plattformbedingte Abbildinkompatibilität entsteht.

Hingegen realisiert die unter [5] vorgestellte Virtualisierungslösung *plattformunabhängiges* Checkpointing auf Byte-Code Ebene. Damit können Abbilder knotenübergreifend genutzt werden. Letzteres gilt jedoch ausschließlich für Anwendungen der OCaml-Sprache.

In Anbetracht bisher existierender Checkpointer, die plattformunabhängige Abbilder generieren, ist unklar, inwieweit diese weiterentwickelt werden, um Unterstützung für *alle Programmiersprachen* anzubieten. Die damit verbundene Notwendigkeit, tiefgreifende Änderungen elementarer Programmiersprachenkonzepte vorzunehmen, lässt dies jedoch als unrealistisch erschienen.

2.3.3 Heterogenität, Effizienz und Virtualisierung

Betriebssystemcontainer, siehe Kapitel 4.3.2, und virtuelle Maschinen isolieren Ausführungsumgebungen verschiedener Benutzer untereinander und sind damit wichtig für Gridumgebungen. Jedoch schlägt die Anwendungsausführung in VMs beispielsweise bei Intel Prozessoren mit 17 Prozent und bei AMD Prozessoren mit 38 Prozent Overhead zu Buche [41][8]. Dies liegt vornehmlich in der virtuellen Speicherverwaltung (TLB, Seitentabellen, Schattenseitentabellen, et cetera) sowie des Netzwerk-, Graphikkarten- und Festplattenzugriffs begründet. Werden VMs gesichert, führt die VM-Abbildgröße, beziehungsweise die damit verbundenen Ein-/Ausgabezeiten zugrundliegender Speichermedien, zu einem großen Checkpointing-Overhead. Es existieren unterschiedliche Techniken zur dynamischen, das heißt *bedarfsgerechten*, Speicherzuweisung an eine VM, siehe [97], [129]. Hierdurch können VM-Abbildgrößen entscheidend verkleinert werden.

2.3.4 Effizienz, Transparenz und anwendungsinternes Wissen

Datenintensive Anwendungen, beispielsweise im Tera-Byte Bereich, müssen und können aus Performanz- und Konsistenzgründen *nicht immer vollständig* gesichert werden. Da ein Checkpointer jedoch über kein anwendungsinternes Wissen verfügt, kann er nicht zwischen sicherungsrelevanten und -irrelevanten Inhalten unterscheiden. Falls Anwendungen abgeändert werden, um bestimmte Inhalte beim Checkpointing auszuschließen, wird das Anwendungstransparenz-Kriterium verletzt.

2.4 Synthese

Die in Unterkapitel 2.3 dargestellten Zusammenhänge bilden die Grundlage für verschiedene Ansätze, Grid-Checkpointing zu realisieren.

[8] Container-basierte Isolierung stellt keine nennenswerten Performanzeinbußen dar [21].

Steht die *Integration heterogener Umgebungen* im Vordergrund, muss mindestens ein Checkpointer pro Gridknoten von der Architektur einbezogen werden können. Wird in diesem Fall *performantes und effizientes Sichern und Wiederherstellen priorisiert*, sollten vorzugsweise Anwendungscheckpointer [9] verwendet werden, als auch verschiedene Sicherungsstrategien umgesetzt werden. Virtual Machines kommen nur in Betracht, wenn viel Arbeitsspeicher und schneller Festplattenspeicher zu Verfügung steht oder bedarfsgerechte Speicherzuweisungen des VMMs an einzelne VMs ermöglicht werden. Im High Performance Computing (HPC) sind Virtuelle Maschinen damit nicht geeignet.

Wird Heterogenität vor Performanz eingestuft, kann jeder Checkpointer, insbesondere VMs, eingesetzt werden, weil hierdurch die meisten Ressourcen gesichert und wiederhergestellt und damit sehr viele Anwendungstypen unterstützt werden können.

Bei einer Grid-Checkpointing-Architektur, in der die Plattformunabhängigkeit des Abbilds im Zentrum steht, kann nur eine Untermenge aller Anwendungen, aufgrund der aktuell existierenden Technologien, unterstützt werden. Im Gegensatz zu vorherigen Architekturen favorisiert diese Architektur den Einsatz eines einheitlichen, beziehungsweise einer geringen Anzahl an Checkpointern.

Ist der Transparenzaspekt maßgeblich, müssen Anwendungen unverändert sicher- und wiederherstellbar sein. Hierzu dürfen System- und Bibliothekcheckpointer, als auch VMs verwendet werden. Anwendungsinternes Wissen kann nicht integriert werden und führt somit unter Umständen zu Sicherungsmehraufwand und erhöhtem Speicherplatzverbrauch.

Bei der Integration heterogener Checkpointer erfordert die Erzeugung konsistenter, verteilter Zustände ein *Zusammenwirken der Checkpointer*, sodass knotenübergreifende Ressourcen korrekt gesichert und wiederhergestellt werden können. In Tabelle 2.1 werden die wichtigsten Entwurfskriterien der Grid-Checkpointing-Architektur von Kapitel 3.1 aufgelistet, welche eine Synthese der unter 2.2 und 2.3 diskutierten Zusammenhänge sind. Der Konflikt von Heterogenität und Skalierbarkeit wird dabei zugunsten von Hetergenität aufgelöst, das heißt, die Integration verschiedenartiger Checkpointerimplementierungen steht im Vordergrund.

[9] Anwendungscheckpointer-produzierte Abbilder sind kleiner als VM-Abbilder, weil sie Systemzustände nur in geringem Maße beinhalten.

Kriterium	Erläuterung
Heterogenität	Heterogene Soft- und Hardwarekomponenten müssen integriert werden.
Konsistenz	Alle Checkpoints müssen konsistent sein.
Transparenz	Anwendungen sollen nicht abgeändert werden müssen, um gesichert oder wiederhergestellt zu werden.
Anwendungsunterstützung	Interaktive, graphische, rechen-, daten- und kommunikationsintensive Anwendungen sollen unterstützt werden.
Effizienz	Der Checkpointingaufwand darf die Systemperformanz und den Anwendungsfortschritt nur begrenzt belasten.
Sicherheit	Anwendungszustände dürfen von Dritten nicht manipuliert, zerstört oder missbraucht werden.

Tabelle 2.1: Elementare Grid-Checkpointing-Kriterien

2.5 Zusammenfassung

Grid-Checkpointing ermöglicht es, Jobs anzuhalten, den Jobzustand zu sichern und die Jobausführung auf den aktuellen oder anderen Gridknoten fortzusetzen. Damit können höherwertige Dienste wie Jobmigration, für die Umsetzung von Lastenverteilung, als auch Jobfehlertoleranz im Grid realisiert werden. Zudem können in der Vergangenheit liegende Jobzustände zu Debuggingzwecken eingesetzt werden.

Grid-Checkpointing wird vorherrschend von den Kriterien Heterogenität, Skalierbarkeit, siehe Kapitel 2.2.4, Transparenz, Anwendungsunterstützung, Effizienz, Bedienbarkeit und Sicherheit geprägt. *Die gleichzeitige Vereinbarkeit all dieser Kriterien ist nicht, beziehungsweise nur unter sehr großem Aufwand möglich.* Einerseits produzieren heterogene Checkpointer heterogene Abbilder. Damit ist Anwendungswiederherstellung daran geknüpft, dass ein abbildkompatibler Checkpointer auf einem Restartgridknoten vorhanden ist, welches die Anwendungswiederherstellung auf eine Gridknotenuntermenge beschränkt.

Andererseits besteht höchste Knoten-Flexibilität, wenn ein einziger Checkpointer verwendet wird. Letzteres ist jedoch unwahrscheinlich in einer Gridumgebung, in der unterschiedliche Organisationen Ressourcen in einer gemeinsamen VO bereitstellen.

Zusätzlich kann anwendungstransparentes Sichern zu erhöhtem Aufwand führen, wenn mehr gesichert wird als aus Anwendungssicht notwendig ist. Wird hingegen anwendungsinternes Wissen in den Sicherungsprozess integriert, bedarf es meist der Anwendungsmodifikation.

Die in dieser Arbeit konzipierte Grid-Checkpointing-Architektur fokussiert sich auf die Heterogenität.

3 Grid-Checkpointing Architektur

In diesem Kapitel wird eine Grid-Checkpointing Architektur (GCA) dargestellt, anhand derer Jobs konsistent, transparent und effizient gesichert und wiederhergestellt werden können. Der Entwurfsschwerpunkt liegt dabei auf *der Integration heterogener Checkpointer*.

Die GCA stellt eine *integrierte Architektur* dar, da sie auf grundlegenden Grid Computing Diensten, siehe Kapitel 1.4, basiert und einzelne, wie Job-Management, um Fehlertoleranz erweitert.

Nach der Darstellung elementarer Architekturkomponenten und Abläufe erfolgt eine *Verfeinerung der Architektur um Teilaspekte*, wie die Einbindung verschiedener Checkpointer, Fehlererkennung, Einbindung fehlertoleranter MPI-Umgebungen und fehlertolerantes Checkpointing. Spezifische Herausforderungen bei Einsatz heterogener Checkpointer werden in den nachfolgenden Kapiteln 4, 5 und 6 detailliert diskutiert.

3.1 Überblick

Das Sichern und Wiederherstellen von Jobs im Grid stellt eine komplexe Aufgabe dar und erfordert, dass verschiedenartige GCA-Komponenten, welche in Abbildung 3.1 vereinfacht dargestellt sind, zusammenarbeiten.

Der verteilte *Job Checkpointer* Dienst ist Hauptbestandteil der GCA. Er fordert ein oder mehrere *Job-Einheit Checkpointer* auf, zum Job gehörende Job-Einheiten zu sichern und wiederherzustellen.

Mithilfe der *Uniformen Checkpointer-Schnittstelle (UCS)* adressiert jeder Job-Einheit Checkpointer einen zugrundeliegenden Checkpointer auf einheitliche Art und Weise. Die UCS wird pro Checkpointer mithilfe einer *Übersetzungsbibliothek (engl. translation library)* realisiert. Sie ermöglicht, die Prozessgruppe einer Job-Einheit anhand eines Checkpointers zu sichern und wiederherzustellen.

Fehlertoleranzmonitore (FT-Monitore) werden vom Job und Job-Einheit Checkpointer zur Aufzeichnung von Prozessabhängigkeiten, der Ermittlung entfernbarer Checkpoints sowie zur Erkennung von Anwendungsfehlern, beziehungsweise zur Erkennung potentiell zur System- und Anwendungs-Beeinträchtigung führender Umstände verwendet, siehe Kapitel 3.3.

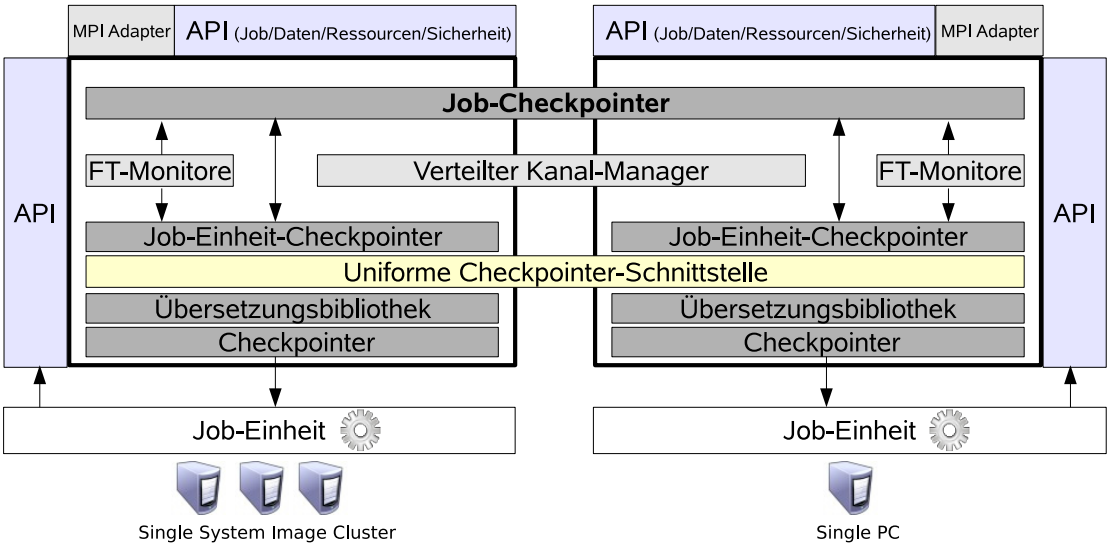

Abbildung 3.1: Grid-Checkpointing-Architekturüberblick

Die GCA muss mit den Diensten des Job-, Daten-, Ressourcen- und Sicherheits-Management kooperieren. Auf elementare Dienste der zugrundeliegenden Grid Computing Architektur greift die GCA über dienstspezifische Schnittstellen zu.

Werden Job-Einheiten, die über Kommunikationskanäle miteinander verbunden sind, zwischen Knoten migriert, beziehungsweise nach einem Fehler auf entfernten Knoten wiederhergestellt, muss der sogenannte Verteilte Kanal-Manager einbezogen werden. Dieser Dienst wird in Kapitel 6 detailliert beschrieben.

Anwendungsinitiierte Sicherungen werden mithilfe einer Bibliotheksschnittstelle zu Diensten der GCA, siehe Kapitel 3.5, ausgeführt.

Mithilfe eines MPI-Adapters können fehlertolerante MPI-Umgebungen in die GCA integriert werden, siehe Kapitel 3.5.2.

3.2 Grundlegende GCA Komponenten

3.2.1 Job-Checkpointer und erweitertes JSDL-Format

Der sogenannte Job-Checkpointer (JC) steuert Job-Sicherungen und -Wiederherstellungen, um Job-Migration, -Fehlertoleranz sowie -Pausierung zu realisieren. Letztere können durch unterschiedliche Akteure *initiiert* werden. Während ein *Grid-Scheduler* Migrationen initiiert, gibt der JC, auf Basis von Fehlermonitorinformationen, den Befehl zur Job-Sicherung, beziehungsweise -Rekonstruktion. Ein *Job-Besitzer* muss sich daher nicht um Fehlerbehandlungen und Rekonfigurationen im Grid-Systems kümmern. Bei Bedarf kann ein Job-

Besitzer jedoch auch über Job-Fehlertoleranz bestimmen und Job-Zustände auf gridinterne und, temporär in das Grid eingeblendete, externe Medien schreiben, oder von diesen lesen. Job-Zustände können hierdurch auch außerhalb des Grids verwaltet werden. Bei der Aus- und Einlagerung müssen jedoch Sicherheitsaspekte und Übertragungszeiten beachtet werden.

Neben Job-Besitzern können *Jobs* Sicherungen initiieren, die vom JC ausgeführt werden. Ein Administrator kann Jobs anhalten, falls beispielsweise eine Knotenwartung notwendig ist. Andererseits kann sie vom Job-Benutzer verwendet werden, um Standby-Funktionalität, insbesondere über Knoten-Reboots hinweg, zu ermöglichen.

Der JC ist eine *Erweiterung des Job-Managers* (JM), siehe Kapitel 1.4.4. Er besitzt dadurch eine globale Sicht auf den Job und kennt somit seine konstituierenden Job-Einheiten sowie deren Ausführungsort.

Ein JC sichert und stellt ausschließlich *Zustände auf Grid-Computing-Ebene* wieder her, dazu zählen alle Strukturen, die einen Job auf Gridebene repräsentieren. Die Zustände der nativen Betriebssystemebene liegen außerhalb seines direkten Zugriffsbereichs. Um die Prozessgruppe einer Job-Einheit zu sichern und wiederherzustellen, *interagiert* der JC mit dem sogenannten Job-Einheit-Checkpointer (JEC), siehe Kapitel 3.2.2.

3.2.1.1 Rollenverteilung bei verschiedenen Checkpointingprotokollen

Die inhärente Grid-Dynamik, verursacht durch unvorhersehbares System-, Netzwerk- und Anwendungsverhalten, muss durch eine Grid-Checkpointing-Architektur reflektiert werden. Hier kann statisch organisierte Fehlertoleranz ineffizient sein. Beispielsweise kann ein zu kurzes Checkpointingintervall bei koordiniertem Checkpointing sehr viel Aufwand im fehlerfreien Betrieb verursachen. Treten keine Fehler ein, so ist ein kurzes Intervall unnötig. Zudem kann intensive Ein-/Ausgabe beim Schreiben und Lesen von Job-Zuständen das *System beeinträchtigen*. Dies trifft insbesondere auf Computersysteme zu, die kein Direct Memory Access (DMA) unterstützen. Beim neueren Busmaster können ein oder mehrere Geräte, die mit dem Bus verbunden sind, Transkationen initieren, welches die Performanz des Betriebssystems steigert. Da Busmaster bei neueren Computersystemen eingesetzt wird, müssen auch ältere Techniken, wie Programed In-/Output (PIO), im Kontext der Hardware-Heterogenitätsunterstützung, integriert werden.

Die Lösung liegt darin, das Sicherungsverhaltens dynamisch anzupassen, beispielsweise indem unterschiedliche Checkpointingprotokolle adaptiv verwendet oder Strategieparameter modifiziert werden. Sich anpassendes Sicherungsverhalten wird unter Kapitel 7 genauer beschrieben.

Das Verhalten des JCs je nach Checkpointingprotokoll wird im Folgenden erläutert.

Koordiniertes Checkpointing:
Der JC übernimmt hierbei die Rolle des Koordinators. Er kommuniziert mit einem oder mehreren JECs, um alle zum Job gehörenden Job-Einheiten zu synchronisieren. Nach-

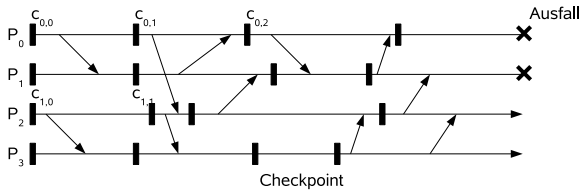

Abbildung 3.2: Checkpoint-Beispielszenario nach [45]

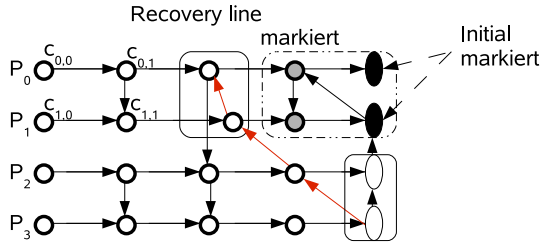

Abbildung 3.3: Rollback-Dependency-Graph nach [45]

dem jeder JEC die Synchronisierung abgeschlossen und eine entsprechende Bestätigung an den JC gesendet hat, beauftragt der JC alle JECs, alle Job-Einheiten zu sichern. Nach jeder Job-Einheitssicherung wird eine Bestätigung an den JC gesendet. Der JC fordert anschließend die JECs auf, die Ausführung der Job-Einheiten fortzusetzen. Auch bei einem koordinierten Job-Neustart steuert der JC die involvierten JECs. Zunächst wird der Wiederaufbau aller zum Job gehörenden Job-Einheiten initiiert. Danach werden alle JECs aufgefordert, die Ausführung der Job-Einheiten fortzusetzen.

Unabhängiges Checkpointing:
Hier *initiiert* der JC die Aufzeichnung von Abhängigkeitsinformationen zwischen den Job-Einheiten während der fehlerfreien Ausführung, sodass alle Job-Einheiten eines Jobs gleichartige Checkpointdaten anlegen.[1] Abbildung 3.2 stellt eine verteilte, über Nachrichten kommunizierende, Beispielanwendung und deren Checkpoints dar.

Beim Restart nimmt der JEC die Rolle des Restart-Initiators ein, der die Wiederherstellungslinie (engl. recovery line) berechnet. Hierzu wird ein sogenannter Rollback-Dependency-Graph [20] erstellt. Hierbei wird eine gerichtete Kante vom Prozesscheckpoint Pi zum Zeitpunkt x ($P_i x$) zum Prozesscheckpoint Pj zum Zeitpunkt y ($P_j y$) gezogen, wenn entweder

1. i ungleich j und eine Nachricht von $P_i x$ gesendet und von $P_j y$ empfangen wurde oder
2. i=j und y=x+1 (gleicher Prozess schickt Nachricht an sich selbst)

[1] Es muss vermieden werden, dass innerhalb eines Jobs für eine Untermenge an Job-Einheiten Abhängigkeitsinformationen aufgezeichnet werden, für die verbleibende Menge hingegen nicht, da sonst inkonsistente Zustände entstehen können.

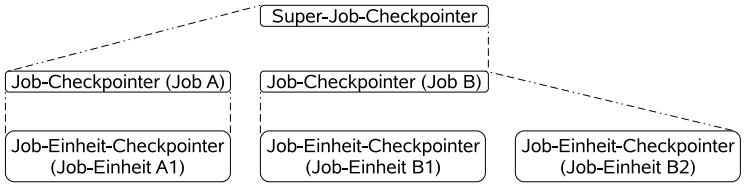

Abbildung 3.4: Super-Job-Checkpointer

gilt [45]. Um Knoten- und Kanteninformationen einzusammeln, sendet der JC eine Abhängigkeits-Anfrage-Nachricht an jeden involvierten JEC, welcher daraufhin die gewünschten Informationen, auf Basis aufgezeichneter Abhängigkeiten, zurückschickt. Wenn der JC, aufgrund der empfangenen Informationen erkennt, dass eine Kante von $P_i x$ nach $P_j y$ existiert und $P_i x$ aufgrund eines Fehlers zurückgerollt werden muss, gilt dies ebenso für $P_j y$. Von den fehlerhaften Knoten rückwärts gehend werden, mithilfe der Erreichbarkeitsanalyse, jene Knoten ermittelt, welche die Recovery-Line darstellen. Abbildung 3.3 verdeutlicht die Berechnung der Wiederherstellungslinie für die Beispielanwendung.

Wurde eine Wiederherstellungslinie erfolgreich berechnet, sendet der JC eine Rollbackanfragenachricht an alle involvierten JECs, welche die Recovery Line und die damit verbundene Checkpointversion einer Job-Einheit beinhaltet. Gegebenenfalls müssen auch fehlerfrei ausgeführte Job-Einheiten auf eine frühere Checkpointversion zurückgerollt werden. Kann kein konsistenter Zustand ermittelt werden, veranlasst er alle involvierten JECs, die jeweilige Job-Einheit zu stoppen und deren initialen Zustand wiederherzustellen.

Unabhängiges Checkpointing mit Nachrichten-Aufzeichnung:
Zusätzlich zur Aufzeichnung von Abhängigkeitsinformationen *initiiert* der JC, dass Determinanten aufgezeichnet werden, was beim senderbasierten (empfängerbasierten) Loggingprotokoll zur Speicherung zu sendender (auszuliefernder) Nachrichten und deren Metadaten führt. Der Restart ist dem unabhängigen Checkpointing sehr ähnlich. Um einen jüngeren Zustand als den mit Checkpoints assoziierten wiederherzustellen, beziehungsweise um einen Rückfall auf den initialen Zustand zu vermeiden, fordert der JC die jeweiligen JECs auf, die aufgezeichneten, nichtdeterministischen Ereignisse wiedereinzuspielen, siehe Kapitel 1.2.3.

3.2.1.2 Job-Abhängigkeiten und Super-Job-Checkpointer

Ein sogenannter Super-Job-Checkpointer (SJC) agiert als dem JC übergeordnete Instanz, siehe Abbildung 3.4 und übernimmt Checkpointingprotokoll-spezifische Aufgaben, wenn Abhängigkeiten zwischen Jobs auftreten.

Eine Gridanwendung kann aus mehreren Jobs bestehen, beispielsweise Workflow-Anwendungen, die aufgrund des Nachrichtenaustauschs Abhängigkeiten untereinander aufweisen. Job-Abhängigkeiten können während der gesamten Job-Lebensdauer bestehen und somit *statischer* Natur sein. Beispielsweise werden alle, zu einer MPI-Welt gehörende, MPI-

Prozesse in einer Datei abgespeichert, die von einer MPI-Umgebung eingelesen wird, um darauf aufbauend die jeweiligen Verbindungen herzustellen. Im Grid-Computing-Bereich können nach [115] Job-Abhängigkeitsinformationen in einer sogenannten Job Submission Description Language (JSDL) Datei abgespeichert und somit im Vorfeld einer Sicherungsaktion identifiziert werden.

Zur Laufzeit können jedoch auch unvorhersehbare, *dynamische* Abhängigkeiten entstehen. Beispielsweise kann ein Hauptspeichersegment von zwei oder mehreren Benutzern gemeinsam genutzt werden, siehe [109]. Um konsistentes koordiniertes Checkpointing durchzuführen, bedarf es deren Abhängigkeitsauflösung zur Laufzeit, siehe Kapitel 3.3.1.[2]

Sind Job-Abhängigkeiten vorab bekannt, wird ein JC per Abstimmung aller JCs zum SJC ernannt. Prozessabhängigkeitsmonitore, siehe Kapitel 3.3.1, überwachen permanent die zwischen Job-Prozessen auftretenden Beziehungen und können entsprechende Abhängigkeitsinformationen, vor allem wenn sie zum Zeitpunkt der Job-Einreichung nicht vorliegen, zur Laufzeit ermitteln. Involvierte JCs können danach Kontakt zueinander aufnehmen und einen SJC bestimmen.

Ein SJC übernimmt ähnliche Aufgaben wie jene eines JCs. Bei koordiniertem Checkpointing wird der SJC zum Koordinator aller involvierter JCs. Hierdurch werden Job-Einheiten in hierarchischer Weise, von der SJC-Ebene über die JC-Ebene hin zur JEC-Ebene, synchronisiert. Die anfallende Netzwerkbelastung wird hierdurch auf mehrere JC-/JEC-Gridknoten, zur verbesserten Skalierbarkeit, verteilt.

Bei unkoordiniertem Checkpointing berechnet der SJC die Recovery-Line, da er alle voneinander abhängigen Jobs kennt. Nachdem alle Abhängigkeiten ermittelt wurden, sendet er Recovery-Anfragenachrichten an die involvierten JCs, welche sie wiederum, angepasst an die jeweiligen JECs, weiterleiten. Gehört der aktuelle Zustand einer Job-Einheit zur Recovery-Line, kann deren Ausführung fortgesetzt werden, andernfalls muss ein JEC zu einem früheren Checkpoint zurückgehen.

3.2.1.3 Metadaten eines Jobs

Zusätzlich zu den, auf unterster Ebene erzeugten, Prozessgruppen-Abbildern und Checkpointprotokoll-abhängigen Daten, wie Abhängigkeits-Informationen und Determinanten, müssen weitere Informationen aufgezeichnet werden. Dies sind sogenannte Grid-Checkpointing-Metadaten, welche abbild- und umgebungsbezogene Informationen enthalten. Diese Daten sind unabdingbar, damit für einen erfolgreichen Job-Restart ein geeigneter Zielgridknoten lokalisiert werden kann. Weiterhin können diese Metadaten für Checkpointingstatistiken und Strategieplanungen verwendet werden.

[2] Bei automatisierter Ermittlung dynamischer Job-Abhängigkeiten, ohne hinreichender Angabe, ab wann eine Abhängigkeit als solche zu interpretieren ist, besteht die potentielle Gefahr, zu viele Jobs zu einer logischen Grid-Anwendung zusammenzufassen. Letzteres kann zur Blockade sehr vieler Jobs, beziehungsweise einer erhöhten Belastung des Systems führen.

Grid-Checkpointing-Metadaten werden unter einem verteilten Grid-Dateisystem für den *ortstransparenten* Zugriff zum Restartzeitpunkt abgespeichert, somit müssen Checkpointdaten im Grid nicht explizit vom Programmierer lokalisiert werden.

Zwei Typen von Grid-Checkpointing-Metadaten existieren, JC-Metadaten und JEC-Metadaten. Letztere werden in Kapitel 3.2.2.2 beschrieben.

Ein JC-Metadatensatz referenziert pro Job-Checkpoint alle involvierten Job-Einheiten, deren IP-Adresse und aktuelle Checkpoint-Versionsnummer. Letztere entspricht nicht immer der Job-Checkpoint-Versionsnummer, beispielsweise können im Kontext unabhängigen Checkpointings einige Job-Einheiten öfter gesichert werden als andere.

Dateiauszug 3.1 stellt einen Beispieldatensatz eines Jobs mit zwei Job-Einheiten dar. Der erste Eintrag entspricht einem koordinierten Checkpoint, weil für beide Job-Einheiten ein Checkpoint, jeweils mit Versionsnummer 0, erzeugt wurde. Im zweiten Eintrag wird ersichtlich, dass nur Job-Einheit A gesichert wurde, beispielsweise bei unkoordiniertem Checkpointing. Der dritte Eintrag entspricht wieder einem koordiniertem Checkpoint, da beide Einheiten gesichert wurden. Die Gridknoten werden jeweils anhand ihrer IP-Adresse identifiziert.

```
0-JobEinheitA-192.168.5.12-V0-JobEinheitB-192.168.5.13-V0
1-JobEinheitA-192.168.5.12-V1-JobEinheitB-192.168.5.13-V0
2-JobEinheitA-192.168.5.12-V2-JobEinheitB-192.168.5.13-V1
```

Listing 3.1: Job-Metadaten (job-metadata.txt)

Bei koordiniertem Checkpointing erzeugt der JC beziehungsweise SJC JC-Metadaten und veranlasst jeden JEC JEC-Metadaten zu extrahieren und abzuspeichern. Bei unkoordiniertem Checkpointing werden beide Metadatentypen vom jeweiligen JEC erzeugt.

3.2.1.4 Sicherheit

Der JC integriert Sicherheitsmechanismen, sodass nur authentifizierte und autorisierte Akteure Job-Sicherungs- und Wiederherstellungsoperationen ausführen dürfen. Ansonsten können Jobs während der Laufzeit durch Checkpointingaktionen unbefugter Dritter beeinträchtigt werden, beispielsweise durch Prozesssynchronisierung bei koordiniertem Checkpointing oder durch Speicherplatzverbrauch, wenn Checkpoints unter dem Benutzerverzeichnis abgelegt werden. Weiterhin enthält ein Checkpoint sensible Benutzerinformationen und Programmzustände, die missbraucht werden können, wenn unbefugte Dritte beim Restart darauf zugreifen. Dieser Missbrauch gelingt vor allem, wenn der Checkpointer zuvor keine Benutzer-gebundenen Sicherheitinformationen in den Checkpoint integriert hat.

Der JC ist dafür verantwortlich, die Integrität jeglicher Checkpointdaten, bei Übertragung zwischen Gridknoten und stationärer Speicherung, zu überwachen.

Weiterhin muss er erkennen, ob ein Zielknoten die gleichen Bibliotheken wie ein Quellknoten bereitstellt, um Anwendungsabstürze nach oder während eines Restarts zu vermeiden. Falls dynamische Bibliotheken nicht im Abbild eingeschlossen sind, können verschiedene, aber gleichnamige Bibliotheken auf einem Zielknoten ein Sicherheitsrisiko darstellen, da hierdurch Schadsoftware in ein Programm eingeschleust werden kann. System-Aktualisierungen, vorgenommen von einem Administrator, müssen von letzterem unterschieden werden können.

3.2.1.5 Freispeichersammlung

Der JC ist verantwortlich dafür, Checkpointdateien zu verwalten. Konkret müssen nicht mehr benötigte Abbilder erkannt und in regelmäßigen Abständen, beziehungsweise wenn der Speicher kanpp wird, entfernt werden. Abbilder, erstellt im Kontext einer Migration, können sofort entfernt werden, nachdem diese beendet wurde. Es ist teilweise sehr komplex entfernbare Abbilder zu erkennen, die im Fehlertoleranz-Kontext erstellt wurden. Hierzu muss mit dem, in Kapitel 3.3.3 beschriebenen, Dienst interagiert werden.

3.2.1.6 Fehlererkennung

Dieser Dienst überwacht den System- und Job-Zustand. Er gibt Empfehlungen an den JC (SJC), welche Sicherungsstrategie anzuwenden ist und zu welchem Zeitpunkt eine Job-Wiederherstellung vorzunehmen ist. Hierfür verwendet der JC den in Kapitel 3.3.2.1 beschriebenen Dienst.

3.2.1.7 Job-Einheit-Positionierung und erweitertes JSDL-Format

Nicht jeder auf einem Gridknoten installierte Checkpointer ist in der Lage eine Job-Einheit konsistent zu sichern, beziehungsweise wiederherzustellen. Derzeit verfügbare Checkpointer sichern und rekonstruieren Softwareressourcen, wie Prozessgruppen, IPC-Objekte, Container-, Datei- und Geräte-Zustände, et cetera in unterschiedlichem Umfang, siehe Kapitel 1.3.1.6. Folglich dürfen Job-Einheiten nur auf Gridknoten platziert werden, deren installierte Checkpointer *alle, von einer Job-Einheit verwendeten, Softwareressourcen*, unterstützen, um Inkonsistenz zu vermeiden. Hierzu müssen die von einem *Checkpointer unterstützten Ressourcen mit den verwendeten Ressourcen einer einzelnen Job-Einheit abgeglichen werden.*

Einerseits kennt die GCA Eigenschaften jedes integrierten Checkpointers. Auf der anderen Seite sind der GCA Ressourcen einer Job-Einheit vor Job-Einreichung nicht bekannt. Für den Abgleich werden daher *anwendungsbeschreibende Informationen benötigt, die in den Prozess der Job-Einreichung integriert werden müssen.*

Im Grid-Computing-Bereich werden Job-Metadaten meist in Form von JSDL-Dateien (OGF-Standard) bei Job-Einreichung angegeben. Mithilfe einer JSDL-Erweiterung können zusätzlich Grid Checkpointing-Informationen eingebunden werden. Diese neuen JSDL-Elemente werden dem Ressourcen-Management-Dienst bei der Job-Einreichung übergeben, um den beschriebenen Abgleich durchzuführen und damit kompatible Knoten zu finden, siehe Kapitel 1.4.3.

Im Dateiauszug 3.2 werden die um Checkpointing-Informationen erweiterte JSDL-Datei des Beispielprogramms dargestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
      <JobCheckpointing>
        <JobCheckpointerMatching>
                <MultiThread>Yes</MultiThread>
                <MultiProcess>Yes</MultiProcess>
                <IPC-shm>Yes</IP-shm>
                <IPC-msgqu>Yes</IP-msgqu>
                <IPC-sem>Yes</IP-sem>
                <Sockets>Yes</Sockets>
                <Pipe>Yes</Pipe>
        </JobCheckpointerMatching>
        <Checkpointer>BLCR</Checkpointer>
        <ProtocolManagement>
                <Name>CoordinatedCheckpointing</Name>
                        <Parameter>1hour</Parameter>
                        <Parameter>Incremental</Parameter>
        </ProtocolManagement>
```

Listing 3.2: Auszug einer, um Checkpointing erweiterten, JSDL-Datei

Die unter dem Tag *JobCheckpointerMatching* angegebenen Felder dienen dazu, die Softwareressourcen einer Anwendung zu spezifizieren. Hierdurch können Knoten mit einem kompatiblen Checkpointer ermittelt werden.

Des Weiteren kann optional mithilfe des Tags *Checkpointer* ein Checkpointer angegeben werden, der aus Programmiererperspektive für die Anwendungssicherung favorisiert wird.

Darüber hinaus können das zu verwendende Checkpointingprotokoll sowie protokollcharakteristische Parameter mithilfe des ProtocolManagement-Tags angegeben werden. Dem Programmierer, beziehungsweise dem Benutzer, wird die Möglichkeit gegeben, die Sicherungsmethodologie zu definieren, weil er, im Gegensatz zum System, die Anwendung kennt. Das System muss keine aufwendige Anwendungsüberwachung vornehmen, um eine Sicherungsstrategie zu bestimmen.

Eine vollständige Auflistung aller Erweiterungstags ist im Anhang unter B.1 zu finden.

3.2.2 Job-Einheit Checkpointer

Der Job-Einheit-Checkpointer (JEC) befindet sich in der GCA unterhalb des JCs. Er wird pro Gridknoten ausgeführt und vom JC adressiert, um eine Job-Einheit, gemäß der selektierten Checkpoint-Strategie, zu sichern und wiederherzustellen. Bei unkoordiniertem Checkpointing entscheidet er autonom über zukünftige Sicherungen.

Als Erweiterung des JEMs, 1.4.7, bildet der JEC die Schnittstelle zwischen Grid-Computing-Software und zugrundeliegendem nativen Betriebssystem. Der JEC besitzt eine Prozesssicht, das heißt, er kennt alle Prozesse, die mit einer Job-Einheit assoziiert werden.

Der JEC verwendet das Schlüsselelement der Grid-Checkpointing-Architektur, die sogenannte *Uniforme Checkpointer-Schnittstelle (UCS)*. Die UCS bindet *mehrere heterogene Checkpointer auf einheitliche Art und Weise* in die GCA ein. Das heißt, dass der JEC nicht abgeändert werden muss, wenn ein neuer Checkpointer in die GCA integriert wird. Anhand der UCS können existierende Checkpointer wiederverwendet werden. Dadurch reduziert sich die Dauer wesentlich, Grid-Fehlertoleranz zu realisieren.

Im Gegensatz zum JC, welcher auschließlich job-bezogene Zustände der Gridebene sichert und wiederherstellt, sichert der JEC Metadaten, die sich auf die von Checkpointern erzeugten Zustandsabbilder der nativen Betriebssystemeebene beziehen. Diese Metadaten werden in Kapitel 3.2.2.2 genauer bechrieben. Der JEC führt keine Operationen auf den elementaren Anwendungsprozessen aus, diese werden nur von den Checkpointern gesichert und wiederhergestellt.

3.2.2.1 Rollenverteilung bei verschiedenen Checkpointingprotokollen

Zur Unterstützung eines adaptiven Sicherungsverhaltens nimmt der JEC bei verschiedenen Checkpointingprotokollen unterschiedliche Aufgaben wahr.

Koordiniertes Checkpointing:
Ein JEC wird vom übergeordneten JC angewiesen werden, Prozessabhängigkeiten von Job-Einheiten verschiedener Jobs zu erkennen und an ihn weiterzureichen, um involvierte Jobs in den Sicherungs- und Wiederherstellungsprozess miteinbeziehen zu können. Während eines Checkpointingvorgangs wird ein JEC vom JC damit beauftragt, alle Prozesse einer Job-Einheit zu synchronisieren, zu sichern und mit deren Ausführung fortzufahren. Die Wiederherstellung einer Job-Einheit, sowie die Fortsetzung der Prozessausführung wird ebenfalls vom übergeordneten JC initiiert.

Unkoordiniertes Checkpointing:
Hierbei ist ein JEC nicht an Checkpointing-Entscheidungen des übergeordneten JCs gebunden, JECs können individuell entscheiden, wann ein günstiger Zeitpunkt zur Job-Einheits-Sicherung ist und diese *autonom durchführen*. Zusätzlich müssen Abhängigkeitsinformationen zwischen Job-Einheiten, während der fehlerfreien Ausführung, transparent für die Anwendung aufgezeichnet werden. Letzteres wird in [48] realisiert, indem relevante Sy-

stemaufrufe mithilfe der Library-Interposition-Technik abgefangen und Abhängigkeitsinformationen vermerkt werden.

Abhängigkeiten zwischen Prozessen *einer* Job-Einheit können an dieser Stelle ignoriert werden, da ein zugrundeliegender Checkpointer den Zustand dieser Prozessgruppe ohnehin sichert.

Beim Restart werden auf Anfrage des JCs Abhängigkeitsinformationen von Job-Einheiten übermittelt. Falls eine Recovery Line vom JC berechnet werden kann, überprüft ein JEC nach Erhalt einer Rollback-Anfragenachricht, ob der gegenwärtige Job-Einheits-Zustand der Recovery Line zugehörig ist. Ist dies der Fall, wird die Ausführung des Prozesses fortgeführt, andernfalls wird die Job-Einheit zu einem früheren Checkpoint, wie in der Recovery Line angegeben, zurückgesetzt. Konnte keine Recovery Line berechnet werden, stoppt der JEC die Job-Einheit-Ausführung und startet die Job-Einheit vom initialen Zustand aus neu.

Unabhängiges Checkpointing mit Nachrichten-Aufzeichnung:
Zusätzlich zu Abhängigkeitsinformationen werden Determinanten transparent für die Anwendung *erfasst*, welches ebenfalls mithilfe der Library-Interposition-Technik umgesetzt werden kann. Der Restart ist dem unabhängigen Checkpointing ohne Nachrichtenaufzeichnung ähnlich. Um einen jüngeren Zustand, als den mit Checkpoints assoziierten wiederherzustellen, beziehungsweise um einen Rückfall auf den initialen Zustand zu vermeiden, realisiert der JEC die Wiedereinspielung nichtdeterministischer, zuvor aufgezeichneter, Ereignisse.

3.2.2.2 Metadaten einer Job-Einheit

Diese Metadaten werden zum Checkpoint-Zeitpunkt ermittelt. Sie werden gespeichert, um einen kompatiblen Gridknoten und Checkpointer für das Abbild einer Job-Einheit zu lokalisieren, sowie notwendige Vorbereitungen durchzuführen, sodass im Fehlerfall ein erfolgreicher Restart gewährleistet werden kann.

Dateiauszug 3.3 stellt beispielhaft einen Ausschnitt relevanter Job-Einheits-Metadaten dar. Die vor Checkpointing verwendeten Hardwareressourcen, wie Speicher, Prozessorkernanzahl, et cetera aufgezeichnet werden, damit gegebenenfalls ein Restartgridknoten lokalisiert werden kann, auf dem dieselben Bedingungen wie auf einem Quellgridknoten bestehen.

Die in Kapitel 3.2.1.7 dargelegte Bindung zwischen Job-Einheit und Checkpointer einerseits, als auch die Tatsache, dass ein Checkpointer an ein natives Betriebssystem gebunden ist, erfordern, dass der Checkpointertyp und dessen -version sowie das Betriebssystem, dessen -version und Wortlänge vermerkt werden.

Mehrere Job-Einheit-Abbilder müssen voneinander unterschieden werden können. Die mit der Job-Einheit assoziierte Prozessgruppen- oder Container-Kennung wird als Checkpointer-Abbild-Kennung verwendet. In Kapitel 4 werden Prozessgruppen und Container ausführlich dargelegt. Weiterhin muss die Version des Checkpointer-Abbildes in den Metadaten gesi-

chert und im Checkpoint-Abbildintegriert werden. Natives BLCR überschreibt beispielsweise permanent das von ihm erzeugte Abbild, sodass eine Unterscheidung mehrerer Abbilder der gleichen Prozessgruppe nicht vorgenommen werden kann.

```
<Hardware>
        <Memory>20MB</Memory>
        <Cpucore>1</Cpucore>
</Hardware>
<Checkpointer>
        <Name>BLCR</Name>
        <Version>0.8.2</Version>
</Checkpointer>
<Operatingsystem>
        <Name>Debian</Name>
        <Version>Lenny</Version>
        <Wordlength>64<Wordlength>
</Operatingsystem>
<Image>
        <Procgrp_cont_type>UNIX-session</Procgrp_cont_type>
        <Procgrp_cont_id>1033</Procgrp_cont_id>
        <Version>1</Version>
</Iimage>
```

Listing 3.3: Job-Einheit-Metadaten (job-unit-metadata.xml)

Wurde bei Job-Einreichung eine Job-Einheit in einem Container gekapselt, müssen Container-spezifische Parameter beziehungsweise Subsysteme vermerkt werden, sodass er beim Restart korrekt wiederaufgebaut werden kann. Container werden in Kapitel 4.3.2 detailliert betrachtet.

Wird die Job-Einheit nicht in einer virtualisierten Umgebung, in einem Container oder einer Virtuellen Maschine ausgeführt, müssen Ressourcenbezeichner von Prozessen, Segmenten, Semaphoren, Nachrichtenwarteschlangen, et cetera aufgezeichnet werden, um Bezeichnerkonflikte vor einem Restart zu erkennen und somit auszuschließen. Ressourcen-Bezeichnerkonflikte entstehen, wenn ein Bezeichner wiederhergestellt werden soll, während er bereits von einem anderen, in der Ausführung befindlichen Prozess, verwendet wird.

Um Sicherheitslücken zu schließen, muss ferner überprüft werden, ob alle von einem Prozess verwendeten Dateien, wie dynamische Bibliotheken oder Datenbanken, insofern sie nicht im Abbild integriert sind, auf dem Zielrechner vorhanden sind. Um Datei-Aktualisierungen erkennen zu können, muss, zusätzlich zum Dateinamen und Dateiversion, die Datei-Prüfsumme ermittelt und aufgezeichnet werden.

Job-Einheit-Abbilder und deren Metadaten werden in einem verteilten Dateisystem zum

ortstransparenten Zugriff beim Restart abgespeichert. Die in Dateiauszug 3.4 angegebene Verzeichnisstruktur, inklusive der job-metadata.txt Datei, ermöglicht, dass mithilfe einer JobID die Metadaten aller involvierten Job-Einheiten eines Jobs und damit die Job-Einheit-Abbilder lokalisiert werden können, ohne komplizierte Abbild-Suchaktionen im Grid einleiten zu müssen.

```
./cp-dir/
        jobID/
                job-metadata.txt
                job-unitID/
                        version/
                                job-unit-metadata.xml
```

Listing 3.4: Verzeichnishierarchie der Metadaten

3.2.3 Uniforme Checkpointer-Schnittstelle

Die Uniforme Checkpointer-Schnittstelle (UCS) bildet das Herzstück der Grid-Checkpointing-Architektur. Der JEC kann mit ihrer Hilfe *auf einheitliche Art und Weise verschiedene zugrundeliegende heterogene Checkpointer adressieren, um Prozessgruppen im Kontext eines ausgewählten Checkpointing-Protokolls zu sichern und wiederherzustellen.* Durch einen einheitlichen Zugriff wird Checkpointertransparenz erzielt, der JEC muss demnach nicht für jedes Checkpointer-Paket angepasst werden.

Die UCS ermöglicht auch nicht-verteilte Checkpointer einzubeziehen, um *verteilte Anwendungen* zu sichern und wiederherzustellen. Hierdurch erfahren diese Checkpointer eine funktionale Aufwertung.

Die UCS muss für jeden Checkpointer, durch eine eigene, sogenannte Übersetzungsbibliothek (engl. translation library), implementiert werden. Spezifisch für jede Übersetzungsbibliothek ist die Auflösung *vertikaler und horizontaler Semantikunterschiede.* Vertikale Semantikunterschiede entstehen, wenn auf einen Gridknoten der JEC und ein zugrunde liegender Checkpointer miteinander interagieren. Horizontale Unterschiede ergeben sich, wenn mehrere Checkpointer, beispielsweise bei der Sicherung und Rekonstruktion von Kommunikationskanal-Zuständen, zusammenarbeiten müssen.

Das Konzept der UCS erfüllt den softwaretechnischen Anspruch der Wiederverwendbarkeit bestehenden Programmtextes, durch Einbindung existierender Checkpointer, in besonderem Maße. Fehlertoleranz im Grid kann hierdurch zeiteffizient realisiert werden.

Neben der Integration von Bibliothekscheckpointern können auch Kernel-Checkpointer über die UCS adressiert werden. Dies ermöglicht einen weitgehenden Zugriff auf Kernel-Ressourcen, sodass *gezielt* Kennungen von Prozessen, Semaphoren, Nachrichtenwarteschlangen, et cetera rekonstruiert werden können, insofern diese Kennungen nicht bereits verwendet werden. Bibliothekscheckpointer sind nicht in der Lage, Ressourcen-Kennungen *gezielt*

wiederherzustellen, aufgrund der fehlenden Schnittstellen zu den entsprechenden Kernfunktionen.

Die UCS wird eingehend in Unterkapitel 3.4 beschrieben.

3.2.4 Klassifikation bestehender Checkpointer

Checkpointer	Ausgewählte Eigenschaften
BLCR	Bibliotheks- und Kernkomponente, Callbacks kein IPC und keine Sockets, keine Container
OpenVZ	viele Kernel-Patches, OpenVZ-Container, deckt fast alle Ressourcen ab, Kanalwiederherstellung basierend auf TCP Session Preservation, keine Callbacks
LinuxSSI	verteilt (SSI), Bibliotheks- und Kernkomponente, cgroup-Container, Callbacks, inkrementelles Sichern
DMTCP	verteilt, Bibliothekscheckpointer, MPI-Unterstützung
zap	Kernkomponente, eigener Containertyp, keine Callbacks
Metacluster	verteilt, eigener Containertyp, proprietär (IBM) deckt fast alle Ressourcen ab
VMWare	Systemcheckpointer (Hardware-Virtualisierung), proprietär, große Abbilder
XEN	Systemcheckpointer (Paravirtualisierung), große Abbilder

Tabelle 3.1: Ausgewählte Checkpointer

Als Checkpointer werden die bereits in Kapitel 1.3 beschriebenen Anwendungs-Checkpointer (Kern- und Bibliotheksebene) und System-Checkpointer bezeichnet. Aufgrund der ständigen Weiter- und Neuentwicklung von Betriebssystemen wird es auch in Zukunft neue Checkpointer(-versionen) geben, die in die GCA eingebunden werden müssen.

Die Tabelle 3.1 enthält ausgewählte Checkpointer mit diversen Eigenschaften.

Die Checkpointer-Vielfalt ergibt sich primär aufgrund:

- unterschiedlicher Ressourcensicherungs- und Wiederherstellungsfähigkeit,
- des Ausmaßes notwendiger Modifizierungen bei Anwendung und/oder Kern,
- unterschiedlicher Optimierungsbestrebungen (inkrementelles und nebenläufiges Sichern),
- der Abhängigkeit vom Betriebssystem bei Checkpointern der Kernebene,
- unterschiedlicher Virtualisierungsansätze und

- kommerzieller Anbieter und der OpenSource-Gemeinde.

Die Implementierung der GCA integriert folgende Checkpointer: BLCR, LinuxSSI, MTCP sowie OpenVZ.

3.3 Erweiterte GCA-Dienste

3.3.1 Monitor zur Ermittlung von Prozessabhängigkeiten

Prozessabhängigkeiten entstehen, wenn mehrere Prozesse miteinander interagieren. Bei koordiniertem Checkpointing müssen alle interagierenden Prozesse, eine Ressource betreffend, ermittelt und synchronisiert werden, damit die Ressource während einer Sicherung nicht durch einen anderen Prozess modifiziert wird. Letzteres führt zu Inkonsistenzen. Implementierungen des unabhängigen Checkpointingprotokolls zeichnen inhärenterweise Abhängigkeiten auf, die durch Kommunikationskanäle entstehen. Sie werden beim Restart verwendet, um einen konsisten Zustand zu ermitteln. Neben Sockets können jedoch auch andere Ressourcen zu Prozessabhängigkeiten führen, beispielsweise:

- Dateien (in den Adressraum normal eingeblendet oder geöffnet),
- gemeinsam genutzte Speichersegmente,
- Nachrichtenwarteschlangen,
- Semaphore und
- Pipes.

Im UNIX/Linux-Bereich kann ein Vater-Prozess das Verhalten seiner Kindprozesse mithilfe von *ptrace* überwachen. Der Vater wird bei jedem Signalempfang des Kindes informiert und kann daraufhin dessen Speicherabbild lesen oder modifizieren. Damit können abhängigkeitsggenerierende Aufrufe identifiziert werden. Da der *init*-Prozess *ptrace* nicht nutzen darf, können insbesondere Prozesse, die ihren Elternprozess verloren haben, nicht beobachtet werden, wodurch Inkonsistenzen entstehen. Zusätzlich unterstützt nicht jeder Kernel-Checkpointer durch *ptrace*-beobachtete Prozesse.

Mithilfe des generischen UNIX/Linux Connector-Frameworks registriert sich ein Überwachungsprozess bei einem sogenannten Connector im Kern, um von spezifischen Kernereignissen über Netlink-Socketkanäle unterrichtet zu werden. Der Process Event Connector (PEC) [69] informiert Überwachungsprozesse über *fork* und *exit* Systemaufrufe. Im Gegensatz zu *ptrace* müssen Agenten in keinem Vater-Kind-Verhältnis zum überwachten Prozess stehen. Connectoren können auf einfache Weise für weitere abhängigkeitserzeugende Systemaufrufe, wie *shmat*(hänge gemeinsam genutztes Segment in Adressraum), *pipe*(erzeuge Interprozesskanal), et cetera erstellt werden. Für Kernel-Checkpointer ist die Connectorbasierte Abhängigkeitsüberwachung transparent. Eine exakte Zuordnung von Semaphoren

und Nachrichtenwarteschlangen zu den nutzenden Prozessen ist jedoch ohne Weiteres im Kern nicht möglich, da Semaphoren- und Warteschlangen konstituierende Kernstrukturen keine Prozessrückverfolgung ermöglichen. Werden die entsprechenden Aufrufe jedoch explizit im Kern abgefangen und aufgezeichnet, können Kernel-Checkpointer autonom, ohne Benutzerinteraktion, die entsprechenden Prozess-Kernstrukturen-Paare bestimmen.

Abhängigkeiten zwischen Prozessen von Job-Einheiten unterschiedlicher Jobs müssen dem Super-Job Checkpointer mitgeteilt werden, sodass die JCs involvierter Jobs adressiert werden können, siehe Kapitel 3.2.1.2.

3.3.2 Monitor für System -und Anwendungsverhalten

Dieser Monitor erkennt Fehler, um eine Wiederherstellung nach einem Fehlerfall initiieren zu können. Der Monitor überwacht weiterhin Anwendungs- und System-Verhalten, um das Checkpointingverhalten adaptiv optimieren zu können.

3.3.2.1 Fehlererkennung

Die GCA maskiert Job-Fehler, die mit dem Fail-Stop Fehlermodell, siehe Kapitel 1.1.2, assoziiert werden. Deshalb steht im Vordergrund, Prozessterminierungen und Knotenabstürze zu erkennen.[3]

Knotenausfälle können mithilfe der sogenannten Herzschlagtechnik (engl. heartbeat, HB) erkannt werden. Solange ein festgelegtes Signal von einem Sender-Knoten innerhalb eines vereinbarten Zeitraums von einem Detektor empfangen wird, gilt der Sender-Knoten als nicht abgestürzt. Bleibt das Signal aus, wird ein Ausfall vermutet und eine übergeordnete Instanz benachrichtigt. Wird der Rechnerzustand jedoch zu häufig ermittelt, kann dies zu Performanzeinbußen führen. Eine Anpassung des Signalintervalls ist somit notwendig.

Zusätzlich müssen Wartezeiten auf HBs angepasst werden, damit unvorhersehbare Netzwerklatenzen fälschlicherweise nicht als Knotenausfall interpretiert werden, obwohl ein HB noch im Netz ist.

In der Literatur existieren enge Verzahnungen von MPI-Anwendungen, Checkpointingprotokollen und Fehlererkennung, siehe [42]. Jedoch sollte eine ideale Fehlererkennungslösung einerseits auf unterschiedliche Anwendungsklassen angewendet werden können, die andererseits unabhängig von *einem* spezifischen Checkpointing-Protokoll ist. Beide Aspekte können mit einer GCA-gestützten Lösung erzielt werden, wobei existierende Dienste und deren Wissen verwendet werden können. Nach [120] wird pro Job-Einheit ein sogenannter HB-Publisher erzeugt, welcher sich auf demselben Knoten wie die zu überwachende

[3] Ein Fehlverhalten des Systems wird bei der Erkennung von Job-Fehlern ausgeschlossen. Beispielsweise könnt die Systemsoftware eines Gridknotens, gemäß dem byzantinischen Fehlermodell, fehlerhafte Falschmeldungen versenden.

Job-Einheit befindet. Ein HB-Publisher sendet HBs an einen sogenannten HB-Listener. Dieser ist auf dem Knoten des Job-Managers, beziehungsweise Job-Checkpointers installiert. Da ein Job-Manager/Checkpointer alle Job-Einheit-Adressen kennt, sind ihm damit alle HB-Publisher-Adressen bekannt. Damit kann er ausbleibende HBs, insbesondere von HB-Publishern abgestürzter Knoten, erkennen.

Weiterhin erkennt ein HB-Publisher, dass ein Anwendungsprozess terminiert hat, falls ein *exit*-Systemaufruf abgefangen und dessen Statuswert positiv auf *EXIT_FAILURE* hin überprüft wurde. Hierfür stehen Techniken wie der Linux' Process-Event-Connector, siehe [69] zur Verfügung. Der Terminierungsgrund kann einem HB hinzugefügt und auf HB-Listener Seite interpretiert werden.

Da alle HBs eines Jobs auf einem Knoten eintreffen, besteht die Gefahr eines *Single Point of Failures*. Die fehlertolerante Ausführung von GCA-Komponenten wird in Kapitel 3.6 beschrieben.

3.3.2.2 Anwendungs- und Systemüberwachung

Checkpointing-Performanz kann in Bezug auf Speicher-Management in vielerlei Hinsicht negativ beeinflusst werden, zum Beispiel durch, parallel zur Checkpoint-Speicherung, ausgeführte Festplatten-Ein-/Ausgabe-Operationen oder Überlastung des Hauptspeichers und Swap-Bereichs. Es kann zum Fehlschlag einer Checkpointingoperation kommen, falls nicht genügend Speicher verfügbar ist. Ein Systemspeichermonitor muss solche Situationen rechtzeitig erkennen und geeignete Maßnahmen initiieren. Beispielsweise können Checkpointingaktionen auf spätere Zeitpunkte verschoben oder quota-Parameter angepasst werden, um Abbilder zu komprimieren oder aus einem zugrundeliegende Griddateisystem auszulagern.

Das Schreib- und Kommunikationsverhalten von Jobs ist bei der Bestimmung der effizientesten Checkpointing-Strategie wichtig und muss in anwendungstransparenter und effizienter Art und Weise durch ensprechende Monitore überwacht werden.

3.3.3 Checkpoint-Verwaltung

Bei effizienter Verwaltung des Festplattenspeicherplatzes müssen entfernbare Abbilder von nicht entfernbaren unterschieden werden. Hierbei muss beachtet werden, dass ein konsistentes logisches Abbild sich aus mehreren Abbilddateien zusammensetzen kann. Da nur die abbilderzeugenden Checkpointer die Abhängigkeiten zwischen Abbilddateien kennen, muss dem JC Zugang zu diesen Informationen über den JEC und einer Schnittstelle zum Checkpointer erhalten, siehe Kapitel 3.4.

Bei unkoordiniertem Checkpointing werden potentiell mehr Abbilder erzeugt, wobei nicht

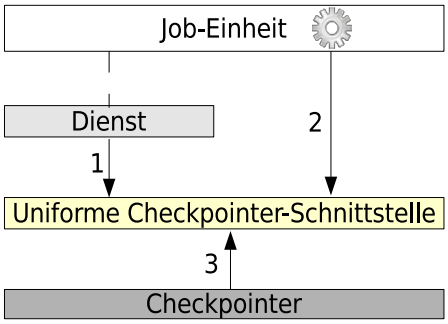

Abbildung 3.5: Komponenten, die den UCS-Entwurf beeinflussen

alle Teil eines konsistenten Zustands sind. Der JC kann mithilfe der Recovery-Line-Berechnung Abbilder bestimmen, die entfernt werden können.

3.4 Uniforme Checkpointer-Schnittstelle

Die Uniforme Checkpointer-Schnittstelle stellt eine einheitliche Schnittstelle zu heterogenen Checkpointern dar. Sie wird im GCA-Kontext von unterschiedlichen Einflussfaktoren geprägt, siehe Kapitel 3.4.1. Das Hauptmerkmal der UCS-Implementierung wird mit dem Begriff *Übersetzung* beschrieben. In Kapitel 3.4.2 werden alle UCS-Übersetzungsdimensionen diskutiert.

3.4.1 Klassifizierung der UCS-Entwurfseinflüsse

Abbildung 3.5 verdeutlicht die Positionierung der Schnittstelle innerhalb der GCA. Sie wird einerseits von funktionalen Anforderungen der Grid-Computing-Dienste und Anwendungen beeinflusst. Andererseits geben Checkpointer-Pakete vor, welche Fehlertoleranz-Funktionalität auf unterster Ebene realisierbar ist.

3.4.1.1 Schnittstellenanforderungen von Grid-Computing-Diensten

Job Submission:
Die erfolgreiche Job-Sicherung und -Rekonstruktion verlangt, dass Job-Einheit und Checkpointer kompatibel zueinander sind, siehe Kapitel 3.2.1 und 3.2.2.2. Daher müssen Informationen, welche ein Checkpointer-Paket beschreiben, von der UCS ermittelt und an höherwertige Grid-Computing Dienste, wie dem Ressource Management, weitergegeben werden können.

In einem sogenannten *Ressourcenkatalog*, der von der Übersetzungsbibliothek des Checkpointers ausgefüllt und an den JC weitergeleitet wird, wird festgehalten, inwiefern folgende Ressourcen und Eigenschaften vom Checkpointer unterstützt werden:

- mehrere Threads (ja)
- mehrere Prozesse (ja)
- IPC shared memory (nein)
- IPC Warteschlangen (nein)
- IPC Semaphore (nein)
- Sockets (ja)
- Pipes (ja)
- reguläre Dateien (ja)
- Pseudodateien (nein)
- Callbacks (nein)
- MPI (nein)
- inkrementelles Checkpointing (nein)

Die unter Kapitel 3.2.1.7 referenzierte Beispielanwendung kann nicht von diesem Checkpointer behandelt werden, weil bei Abgleich der JobCheckpointMatching-Tags der JSDL-Datei und der Felder des Ressourcenkatalogs keine Übereinstimmung bezüglich der Inter-Process-Communication (IPC)-Objekte erzielt werden konnte.

Grid-Dienste, die auf Checkpointing aufbauen:
Migration kombiniert koordiniertes Job-Checkpointing auf Quellgridknoten und koordinierten Job-Restart auf Zielgridknoten. Bei Fehlertoleranz müssen vorrangig Details der jeweiligen Checkpointingstrategie als Parameter übergeben werden können, beispielsweise Checkpointing-Intervall, Strategie-Protokoll, Replikations-Level der Abbilder, et cetera.

Grid-Sicherheit:
Die USC muss gewährleisten, dass Checkpointing nur von authentisierten und autorisierten Benutzern auf Prozessgruppen durchgeführt werden darf. Die Herausforderung hierbei liegt im Identitätsmanagement. UCS muss absichern, dass die Abbildung eines globalen Gridbenutzers auf denselben lokalen Benutzer auf jedem Gridknoten vorliegt, um unbefugte Dritte von Sicherungs- und Fehlertoleranzoperationen auszuschließen.

Freispeichersammlung (Garbage Collection):
Insbesondere voneinander abhängige Checkpointabbilder, beispielsweise inkrementelle Prozessgruppenabbilder, dürfen im Zuge der Freispeicherverwaltung nicht ohne weiteres entfernt werden, weil dadurch Datenkonsistenz gefährdet ist. Da in den meisten Fällen abhängige Abbilder nur dem Checkpointer bekannt sind, muss einem Garbage Collection-Dienst

höherer Ebene anhand der UCS Zugang zu diesen Informationen ermöglicht werden, siehe Kapitel 3.3.3.

3.4.1.2 Schnittstellenanforderungen von Job-Einheiten

Einbezug benutzerdefinierter Funktionen:
Einer Anwendung muss die Möglichkeit gegeben werden, benutzerdefinierte Funktionen einem Sicherungs- beziehungsweise einem Wiederherstellungsprozess hinzuzufügen, um einerseits fehlende Sicherungs- beziehungsweise Wiederherstellungsfähigkeiten einzelner Checkpointer auszugleichen. Andererseits muss anwendungsinternes Wissens einbezogen werden können, um nur jene Ressourcen zu sichern, die aus Sicht der Anwendung sicherungsrelevant sind.

Kooperation beim Sichern gemeinsamer Ressourcen:
Sind gemeinsam genutzte Ressourcen, wie Kommunikationskanalzustände, zu sichern und wiederherzustellen, ist eine explizite Kooperation heterogener und/oder homogener Checkpointer notwendig, da ein Checkpointer in der Regel keinen anderen Checkpointer kennt. Eine Kooperation kann durch Checkpointermodifikationen erreicht werden, ist jedoch nicht praktikabel. Deshalb muss die UCS einen Mechanismus integrieren, anhand dessen beispielsweise Kanalzustände in einheitlicher Form gesichert und wiederhergestellt werden können, ohne Checkpointer und Anwendungen zu modifizieren.

3.4.1.3 Schnittstellenanforderungen bei Integration nativer, heterogener Checkpointer

Koordiniertes Checkpointing:
Soll ein koordinierter Checkpoint, beziehungsweise ein Restart durchgeführt werden, können keine Checkpointer-nativen Kommandos verwendet werden. Anstelle der üblichen *checkpoint-* und *restart*-Sequenzen müssen Teilsequenzen anhand der UCS angesteuerbar sein, um konsistente Abbilder bei verteilten Anwendungen zu generieren. Zwei Beispiele verdeutlichen diesen Aspekt.

Beim koordinierten Sichern zweier Prozesse, die einerseits jeweils zu unterschiedlichen Job-Einheiten gehören und andererseits eine Datei des Grid-Dateisystems jeweils in ein Speichersegment einblenden, müssen zwei Checkpointer beide Prozesse zunächst schlafen legen, bevor sie die Job-Einheiten sichern können. Ein inkonsistentes Abbild entsteht, wenn Prozess P1 die gemeinsame Datei zu Zeitpunkt t1 sichert, Prozess P2 diese noch mehrfach modifiziert und erst später, zu Zeitpunkt t2, sichert. Zum Restartzeitpunkt wird einer der beiden Prozesse einen Inhalt vorfinden, der sich von dem des Checkpointzeitpunktes unterscheidet. Das Kriterium, dass alle Leseoperationen den global zuletzt geschriebenen sehen, kann nicht eingehalten werden. P1 kann erst nach Checkpointbeendigung lesen, zwischenzeitliche Dateiänderungen, seitens P2, werden P1, bei fehlender Synchronisierung,

vorenthalten.

Analog dazu muss die UCS auch bei koordiniertem Restart eine gemeinsame und schrittweise Zusammenarbeit der Checkpointer ermöglichen. Dauert beispielsweise die Wiederherstellung einer Job-Einheit kürzer als die der anderen Job-Einheiten einer verteilten Anwendung, arbeitet die bereits wiederhergestellte Job-Einheit auf Daten, die Job-Einheiten, welche noch wiederhergestellt werden, niemals sehen werden.

Unkoordiniertes Checkpointing/Restart:
Für die Recovery-Line-Berechnung müssen zunächst aufgezeichnete Checkpoint-Abhängigkeitsinformationen vom JC angefordert werden. Anschließend werden die Recovery-Line-Informationen des JC's über die USC an die Übersetzungsbibliothek weitergeleitet.

Unkoordiniertes Checkpointing mit Nachrichten-Aufzeichnung:
Zusätzlich zu den im vorangehenden Abschnitt erwähnten Aspekten muss die UCS ermöglichen, dass nichtdeterministische Ereignisse aufgezeichnet werden.

Einstellbarkeit von Strategieoptionen:
Damit unterschiedliche Sicherungsmethoden ausgeführt werden können, müssen verschiedene Sicherungsoptionen, wie inkrementelles, nebenläufiges oder repliziertes Sichern et cetera einstellbar sein, um individuelle Fähigkeiten verschiedener Checkpointer auszuschöpfen.

Checkpointer-Registrierung:
Die UCS-Implementierung registriert einen Checkpointer beim JEC unter Angabe der Checkpointer-spezifischen Sicherungs- und Wiederherstellungsfähigkeiten. Die UCS-Implementierung füllt dabei den vom JEC eingereichten Ressourcenkatalog, siehe Kapitel 3.4.1.1, aus.

Auflösung von Abbildabhängigkeiten:
Ein Checkpointer stellt Informationen über Abbildabhängigkeiten höhergelagerten Diensten, wie dem Checkpoint-Verwaltungs-Dienst, siehe Kapitel 3.3.3, über eine weitere Schnittstelle zur Verfügung.

3.4.2 UCS-Übersetzungsdimensionen

Die in Kapitel 3.4.1 aufgelisteten Anforderungen beinhalten horizontale (Checkpointer-zu-Checkpointer) und vertikale (Checkpointer-zu-JEC) Semantikunterschiede, welche von der UCS aufgelöst werden müssen.

Grid-Semantiken versus Betriebssystem-native Semantiken (vertikal):
Das native Betriebssystem kennt das darüberliegende Grid-Computing System nicht. Letzteres wiederum besitzt nur ein eingeschränktes Wissen eines zugrundeliegenden nativen Betriebssystems. Hierbei treten unterschiedliche Semantiken auf, welche beide Schichten verwenden. Ein Job, als Ausführungseinheit der Gridebene, wird anders verwaltet als ein Prozess der nativen Betriebssystemebene. Es muss abgesichert werden, dass ein Checkpointer nur die zu einer Job-Einheit gehörenden Prozesse adressiert, siehe Kapitel 4.1.

Weiterhin müssen die Identitäten eines Benutzers auf Grid- und auf nativer Betriebssystemebene korrekt aufeinander abgebildet werden, damit die Sicherheitsmechanismen beider Ebenen konfliktfrei und korrekt ausgeführt werden. Die UCS muss daher entsprechende Informationen zur korrekten Identitätsabbildung weiterleiten.

Aufrufsemantiken (vertikal):
Die UCS muss die einheitlichen JEC-Befehle zur Sicherung beziehungsweise Wiederherstellung einer Job-Einheit in den Checkpointprotokoll-spezifischen Einstiegspunkt bei einem Checkpointers übersetzen. Ein Einstiegspunkt entspricht dem Beginn eine Teil-Sequenz, welche unter 3.4.4 detailliert beschrieben werden.

Checkpointer (vertikal):
Die UCS muss einerseits unterschiedliche Checkpointertypen, andererseits verschiedene Versionen des gleichen Types unterstützen. Betriebssysteme verändern sich über die Zeit hinweg. Jede neue beziehungsweise aktualisierte und zu sichernde Kernstruktur wirkt sich auf das Abbildformat aus und beeinflusst damit die Abbild-Interpretationsfähigkeit eines Checkpointers.

Callbacks (vertikal):
Jeder Checkpointer, der Callbacks unterstützt, stellt eine eigene Bibliothek mit individuellen Callback-Registrierungsroutinen bereit. Weil Anwendungsprogrammierer die zukünftig mit der Anwendung assoziierten Checkpointer nicht kennen, werden Callbacks anhand einer einheitlichen Callbackschnittstelle registriert. Erst nachdem der mit der Anwendung assoziierte Checkpointer bekannt ist, wird die Checkpointer-spezifische Callback-Registrierung vorgenommen. Dieses Vorgehen verhindert eine feste Verdrahtung von Anwendung und Checkpointer und schränkt damit die zur Verfügung stehenden Knoten nicht ein.[4]

Checkpointer-Kooperationen (horizontal): Verwenden zwei oder mehr Job-Einheiten eine gemeinsame Ressource, wie beispielsweise Dateien, Pipes oder Kommunikationskanäle, müssen zwei oder mehr Checkpointer miteinander kooperieren, damit konsistente Job-Sicherungen und -Wiederherstellungen realisiert werden. Semantikunterschiede zwischen Checkpointern, beispielsweise unterschiedliche Markernachrichten zum Leeren eines Kommunikations-Kanals, müssen hierbei von der UCS berücksichtigt werden.

3.4.3 Logische Bestandteile der UCS

Nachdem diverse UCS-Anforderungen diskutiert worden sind, ergibt sich eine erste Struktur der benötigten UCS-Komponenten. Abbildung 3.6 verdeutlicht die notwendige Auftei-

[4] Diese Methode darf jedoch nur in Erwägung gezogen werden, wenn Callback-Funktionen keinen Checkpointer-spezifischen Code enthalten. BLCR erfordert beispielsweise die Integration des cr_checkpoint Aufrufs, um zwischen Zeitpunkten der Callback-Ausführung (vor oder nach einem Checkpoint, nach einem Restart) unterscheiden zu können. Callback-Einbindung bei BLCR setzt voraus, dass die Anwendung statisch gegen die BLCR Bibliothek gelinkt wird. Andere Callback-unterstützende Checkpointer wie LinuxSSI besitzen ähnliche Bedingungen.

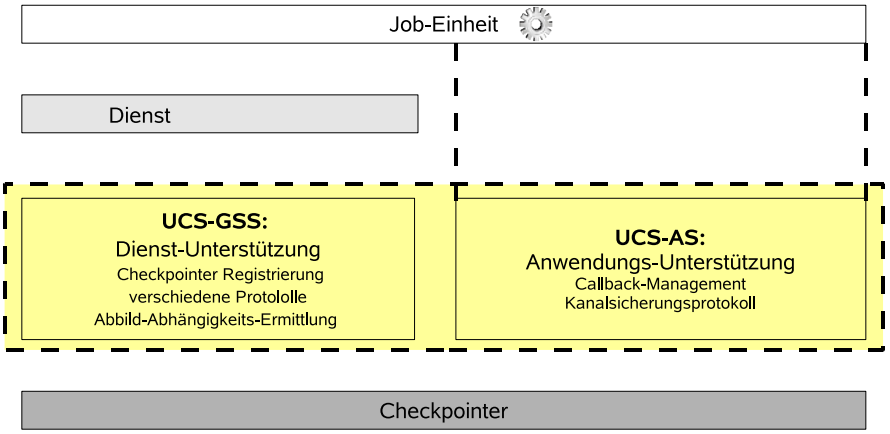

Abbildung 3.6: Logische Bestandteile der UCS

lung der UCS in zwei logische Teilkomponenten.

UCS-AS (Anwendungsunterstützung) entspricht jener UCS-Funktionalität, welche *im Adressraum der Anwendung* ausgeführt wird. Hingegen stellt USC-GSS (Griddienstunterstützung) die *für die GCA-Dienste* notwendigen Funktionalitäten bereit. Sie werden außerhalb des Anwendungsadressraums ausgeführt.

UCS-AS- und UCS-GSS-Funktionalitäten sind disjunkt. Die UCS-AS-Realisierung wird im Folgenden genauer beschrieben.

3.4.4 Funktionen der UCS

3.4.4.1 Checkpointer-Registrierung

register_checkpointer(ressource_catalog)

Dieser Aufruf erfolgt anhand der UCS-GSS unmittelbar nach dem Systemstart, beziehungsweise nach Installation einer neuen Übersetzungsbibliothek. Die Funktion füllt den unter Kapitel 3.4.1.1 beschriebenen Ressourcenkatalog aus. Dabei werden gleichzeitig Informationen über die vom Checkpointer unterstützten Prozessgruppen und/oder Container ausgetauscht. Diese Information ist notwendig bei der Zuordnung einer Job-Einheit zu einem Gridknoten. Zusätzlich können Checkpointer-individuelle Vorbereitungen getroffen werden, beispielsweise indem eine Job-Einheit in einem Container gekapselt oder eine spezifische Prozessgruppe angelegt wird.

API-Name	Kontext
register_checkpointer	Ressourcen-Abgleich
stop_jobunit	CP
checkpoint_jobunit_coordinated	CP
resume_jobunit_cpcontext	CP
record_process_dependencies	CP
stop_process_dependency_recording	CP
record_process_dependencies_and_nondeterministic_events	CP
checkpoint_jobunit_independent	CP
rebuild_jobunit	RST
resume_jobunit_rstcontext	RST
get_process_dependencies	RST
restart_jobunit_replay_logs	RST
register_callback	Callback Management
deregister_callback	Callback Management
return_incremental_image_file_dependencies	Abbild Management

Tabelle 3.2: UCS-Überblick (CP=Checkpoint, RST=Restart)

3.4.4.2 Teilsequenzen des koordinierten Checkpointings

Das koordinierte Checkpointing-Protokoll kann anhand von fünf generischen Aufrufen, drei für die Sicherung und zwei für die Wiederherstellung, für heterogene Checkpointer realisiert werden.

stop_jobunit(stop_info)

Dieser Aufruf bündelt die Ausführung registrierter Pre-Checkpoint-Callback-Funktionen, siehe Kapitel 5, und anschließende Synchronisierung aller Job-Einheitsprozesse.

Wie in Kapitel 3.4.1.3 dargestellt, muss die Checkpointer-native Sicherungs-Sequenz aufgeteilt werden. Teilsequenzen sind die Prozessgruppensynchronisierung, -Sicherung und -Aufweckung. Um diese Teilsequenzen schrittweise auszuführen, werden Unterbrechungen an zwei Stellen in der Checkpointer-nativen Sicherungs- und an einer Stelle der Wiederherstellungssequenz integriert. Damit die Übersetzungsbibliothek gezielt Teilsequenzen in einem Bibliothekscheckpointer ansteuern kann, werden beidseitig Nachrichten über eine Nachrichtenwarteschlange ausgetauscht. Erst wenn eine bestimmte Signalnachricht eingetroffen ist, erfolgt der nächste Schritt im Checkpointer oder der Übersetzungsbibliothek.

Um eine Checkpointer-native Sicherungssequenz aufzuschlüsseln, muss gegebenenfalls die Threadanzahl pro Anwendungsprozess ermittelt werden, damit erst dann mit der Sicherung begonnen wird, nachdem *alle* Threads deren registrierte Callbacks ausgeführt haben und synchronisiert wurden. Die Anwendungsberechnung darf anschließend erst dann fortgeführt werden, nachdem alle Prozesse gesichert wurden. Beim Restart müssen die Kernstruktu-

ren aller Threads zunächst wiederaufgebaut werden, bevor Threads dem Prozesor zugeteilt werden können.

Damit eine Übersetzungsbibliothek gezielt Teilsequenzen im Kern ansteuern kann, wird der *ioctl*[5]-Systemaufruf verwendet. Beispielsweise wurde LinuxSSI um Teilsequenzen erweitert, die über *ioctl* referenziert werden können.

checkpoint_jobunit_coordinated(checkpoint_info)

Dieser Aufruf erzeugt ein Prozessgruppenabbild und kann in unterschiedlichen Kontexten initiiert werden. Bei migrations-bedingtem Sichern muss ein Prozessgruppenabbild temporär zwischengespeichert und unmittelbar nach Wiederherstellung wieder gelöscht werden. Bei Fehlertoleranz müssen die erzeugten Abbilder längerfristig aufgehoben werden, bevor sie wieder entfernt werden können. Die von einem Benutzer initiierten Abbilder werden unter einem Benutzer-Verzeichnis abgelegt.

Über einen zusätzlichen Parameter müssen *unterschiedliche Checkpointstrategien einstellbar* sein. Beispielsweise müssen Ausführungsdatei, Bibliotheken, geöffneten Dateien, et cetera selektiv gesichert werden können, um Sicherungsaufwand zu reduzieren. Inkrementelles, nebenläufiges oder vollständiges Sichern entspricht weiteren Strategieparametern, die anhand des checkpoint_info Parameters eingestellt werden. Diese Checkpointing-Parameterisierung bildet die Voraussetzung für adaptives Checkpointing.

Ein weiterer Parameter referenziert den Speicherort von Abbildern. Im Allgemeinen legen Checkpointer Abbilder auf lokaler Festplatte ab oder speichern sie temporär im lokalen oder entfernten Hauptspeicher, siehe Ghost-Prinzip Kerrighed [144]. Hingegen können Abbilder persistent und ortstransparent, beispielsweise anhand eines Griddateisystems, aufbewahrt werden. Hierbei werden Abbilder zum Zeitpunkt der Job-Wiederherstellung implizit vom Dateisystem lokalisiert. Es ist kein zusätzlicher Mechanismus', wie beispielsweise ein verteilter Verzeichnisdienst notwendig.

Die Ausfallsicherheit von Dateiinhalten kann mit Replikation erzielt werden und wird über einen weiteren Parameter gesteuert. Die Angabe ist jedoch an die Fähigkeiten des zugrundeliegenden Dateisystems, beispielsweise XtreemFS oder eines anderen Dateimanagements wie gridFTP, gebunden.

resume_jobunit_cpcontext(resume_cp_info)

Dieser Aufruf vollendet einen Sicherungsvorgang, indem alle Prozesse einer Prozessgruppe aufgeweckt, beziehungsweise in den Bereit-Zustand versetzt werden. Bei Job-Einheiten, deren Sicherungsvorgang kürzer ist als der anderer Job-Einheiten, können vorzeitig gemeinsam genutzte Ressourcen modifiziert werden, sodass in der Sicherung befindliche Job-Einheiten diese Änderungen nicht erkennen können. Dieser Aufruf trägt wesentlich dazu bei, strikte Konsistenz von Anwendungsdaten, über Sicherungen hinaus, beizubehalten. Der Aufruf beinhaltet zudem die Ausführung von Post-Checkpoint-Callback-Funktionen,

[5]Dieser Aufruf ermöglicht die Kommunikation eines Prozess mit einer Gerätedatei und dient damit als Schnittstelle zwischen Benutzeradressraum und Kern.

siehe Kapitel 5.

rebuild_jobunit(rebuild_info)

Dieser Aufruf rekonstruiert eine Prozessgruppe unter Verwendung eines Checkpointabbilds. Da die Wiederherstellung einzelner Job-Einheiten unvorhersehbar ist, wird keine von ihnen in den Bereit-Zustand versetzt, analog zum **checkpoint_jobunit_coordinated** Aufruf.

resume_jobunit_rstcontext(resume_rst_info)

Dieser Aufruf versetzt die wiederhergestellten Job-Einheitsprozesse in den Bereit-Zustand. Wird den Prozessen die CPU zugeteilt, erfolgt die Ausführung der Restart-Callback-Funktionen, siehe Kapitel 5.

3.4.4.3 Teilsequenzen des unkoordinierten Checkpointings

Im Grid-Kontext bezieht sich ein unkoordinierter Sicherungsvorgang auf *alle* Prozesse *einer* Job-Einheit, weil in der Praxis ein Checkpointer auf Prozessgruppengranularität sichert und wiederherstellt. Hingegen wird unkoordiniertes Checkpointing in der Theorie mit der Sicherung eines einzelnen Prozesses zu einem Zeitpunkt assoziiert.

record_process_dependencies()

Dieser Aufruf initiiert die Aufzeichnung von Prozessabhängigkeiten. Je nach Implementierung kann ein Kernel-Checkpointer gezwungen werden, abhängigkeitserzeugende Aufrufe im Kern selber zu ermitteln. Andernfalls können relevante Systemaufrufe auf Benutzerebene mithilfe der sogenannten Library-Interposition-Technik, siehe Kapitel 5.3.2, abgefangen und Abhängigkeiten aufgezeichnet werden.

stop_process_dependency_recording()

Dieser Aufruf beendet die Aufzeichnung und wird verwendet, um zwischen verschiedenen Protokollen zu wechseln.

checkpoint_jobunit_independent(checkpoint_independent_info)

Mit diesem Aufruf wird genau eine Prozessgruppe synchronisiert, gesichert und erneut in den Bereit-Zustand versetzt. Existierende Abhängigkeiten zu anderen Prozess(-gruppen) werden hierbei nicht aufgelöst. Es gelten dieselben Parameter wie beim Sicherungsaufruf koordinierten Checkpointings.

get_process_dependencies()

Der Aufruf wird indirekt vom JC ausgeführt. Hierdurch werden ihm die aufgezeichneten Abhängigkeiten bereitgestellt, sodass die Recovery Line-Berechnung vorgenommen werden kann.

restart_jobunit(restart_info)

Mit diesem Aufruf wird mitgeteilt, ob eine Prozessgruppe zurückgerollt, weiter ausgeführt

oder vom initialen Zustand neu gestartet werden muss.

3.4.4.4 Unkoordiniertes Checkpointing mit Nachrichtenaufzeichnung

record_process_dependencies_and_nondeterministic_events()

Dieser Aufruf wird benötigt, um zukünftig nichtdeterministische Ereignisse aufzuzeichnen. Analog zu Kapitel 3.4.4.3 wird hierbei entweder der Checkpointer oder die Wrapper-Bibliothek adressiert.[6]

restart_jobunit_replay_logs(restart_replay_info)

Dieser Aufruf rekonstruiert eine Job-Einheit und initiiert die Wiedereinspielung seit dem letzten Checkpointing gepufferte Ereignisse, bevor neue auftreten können.

3.4.4.5 Callback-Management

Die UCS-AS stellt eine einheitliche Schnittstelle bereit, um Callback-Funktionen von Anwendungen registrieren zu können, obwohl der zukünftig mit der Prozessgruppe assoziierte Checkpointer noch unbekannt ist. Erst nach der Zuweisung zu einem Checkpointer sorgt die UCS-AS-Implementierung dafür, dass Callbacks bei der Callback-Infrastruktur des Checkpointers registriert werden, falls diese vorhanden ist. Alternativ bietet die UCS-AS-Implementierung eine eigene Callback-Infrastruktur an, um Checkpointer zu integrieren, die keine Callbacks anbieten.

register_callback(hook, cbfunc, args, migftcontext)

Diesem Aufruf wird der Callbackausführungszeitpunkt (Pre-Checkpoint, Post-Checkpoint, Restart), eine Funktionsreferenz und der intendierte Ausführungskontext des Callbacks, Signalhandler- oder Threadkontext, siehe Kapitel 5, übergeben. Wird die UCS-AS-Callback-Implementierung verwendet, muss zusätzlich eine Kommunikationstechnik vereinbart werden, anhand derer die Callbackausführung von der Übersetzungsbibliothek aus initiiert werden kann. Signale sind in den meisten Fällen bereits belegt, daher können Nachrichtenwarteschlangen, UNIX Domain Sockets oder Pipes eingesetzt werden.

deregister_callback(cb_info)

Hierdurch können Callbacks abgemeldet werden.

[6] Analog zu oben wird der Aufruf *stop_process_dependencies_and_nondeterministic_events_recording* für den Wechsel zwischen Protokollen verwendet.

3.4.4.6 Abbildabhängigkeiten

Da ein Garbage Collection Dienst kein, beziehungsweise nur eingeschränktes Wissen um voneinander abhängige Abbilddateien besitzt, können konsistente Zustände zerstört werden.

Konsistente Abbilder, die im Kontext unkoordinierten Checkpointings erstellt worden sind, werden unter Zuhilfenahme unterschiedlicher Abhängigkeitsgraphen und der Recovery Line Berechnungen ermittelt [12] und [37].

return_incremental_image_file_dependencies(dependency_info)

Mit diesem Aufruf werden Abhängigkeitsinformationen inkrementeller Abbilder von zugrundeliegenden Checkpointern an höherwertige Dienste weitergeleitet. Bei inkrementellem Checkpointing nach 7.5 müssen beispielsweise Einträge der Checkpointer-internen Kontrollstruktur extrahiert werden.

3.5 Erweitertes Fehlertoleranz-Management

Bisher wurde ausschließlich GCA-initiiertes Fehlertoleranz-Management betrachtet. Nun wird die Job-Einheit selber in die Lage versetzt, Fehlertoleranz in unterschiedlichen Formen zu beeinflussen, beispielweise:

1. mit anwendungsdefiniertem Checkpointing,
2. mit anwendungsinitiiertem Checkpointing oder
3. mit einer GCA-externen MPI-Umgebung.

Anwendungsdefiniertes Checkpointing bedeutet, Callback-Funktionen einzusetzen, siehe Kapitel 5. Anwendungsentwickler kennen die Charakteristiken ihrer Anwendungen. Daher definieren sie gezielt jene Callbacks, mit denen nur Inhalte gesichert werden, die tatsächlich gesichert werden müssen. Zusätzlich können sie logische, für eine Anwendung optimale, Zeitpunkte für eine Sicherung angeben. Mithilfe von Callback-Funktionen kann die Anwendung den Sicherungsvorgang optimieren und/oder komplett übernehmen und unabhängig von einem zugrundeliegenden Checkpointer-Paket, der in generischer Weise sichert, agieren.

Bei anwendungsinitiiertem Checkpointing werden anwendungsüberwachende und -sichernde GCA-Dienste von der Job-Einheit *verwendet*, sie werden also nicht vom Anwendungsentwickler definiert und pro Anwendung entwickelt. Anwendungs-definiertes und -initiiertes Fehlertoleranz-Management werden unter Kapitel 3.5.1 näher betrachtet.

Der dritte Aspekt bezieht sich auf der Koexistenz von GCA und einer externen Fehlertoleranz-Umgebung, die einerseits nicht anhand einer Übersetzungsbibliothek in die GCA integriert und andererseits nicht von der Anwendung separiert werden kann. Die Kooperation von MPI-Umgebungen und GCA wird in Kapitel 3.5.2 betrachtet.

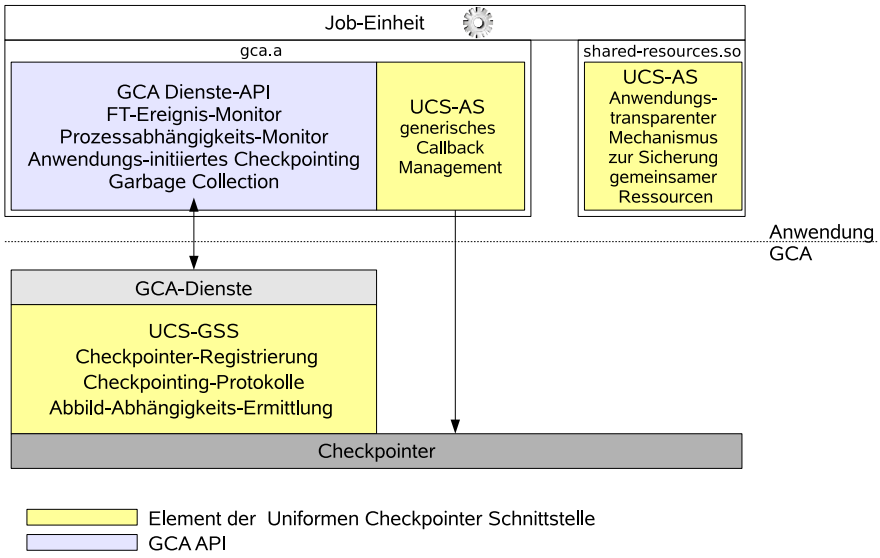

Abbildung 3.7: GCA-Anwendungsbibliotheken

3.5.1 Fehlertoleranz-Management der Anwendung

Abbildung 3.7 stellt zwei Bibliotheken dar, anhand derer anwendungsdefiniertes und anwendungsinitiiertes Checkpointing ermöglicht wird. Mithilfe der *gca.a* Bibliothek kann eine Job-Einheit auf elementare GCA-Dienste zugreifen. Beispielsweise können die in Kapitel 3.3 dargestellten GCA-Monitore adressiert werden, um Informationen über Fehler und Prozessabhängigkeiten zu ermitteln. Darauf aufbauend kann die Job-Einheit Checkpoints initiieren und erzeugte Abbilder verwalten. Der UCS-AS-Teil der *gca.a* Bibliothek registriert zudem Callbackfunktionen, die von Anwendungsprogrammierern entwickelt wurden.

Damit eine Anwendung die beschriebenen Dienste in Anspruch nehmen kann, muss sie die statische *gca.a*-Bibliothek in ihrem Quelltext integrieren, welche die Dienstschnittstelle implementiert. Hierzu muss die Anwendung modifiziert und neukompiliert werden.

Die *shared-ressources.so* Bibliothek unterstützt Fehlertoleranz von Ressourcen, welche von mehreren Job-Einheiten verwendet werden und daher miteinander kooperierender Checkpointer bedarf. Neben Kanalzuständen werden hierdurch Griddateisystem-Dateien konsistent gesichert und wiederhergestellt, indem zwischen den involvierten Checkpointern verhandelt und koordiniert wird. Die Funktionalität der *shared-ressources.so* Bibliothek wird transparent in die Anwendung eingebunden mithilfe der Library-Interposition-Technik und Wrapper-Funktionen, siehe Kapitel 5. Daher wurde diese Bibliothek als dynamisch (vor)ladbare Bibliothek (shared object, .so) realisiert.

3.5.2 MPI-Integration

In Kapitel 1.3.3 wurden existierende MPI-Implementierungen dargestellt, die verteilte Fehlertoleranz, unabhängig von der GCA, realisieren. Da viele der heutigen wissenschaftlichen Anwendungen auf MPI basieren, ist es wichtig, MPI-Anwendungen fehlertolerant in Grid-Umgebungen ausführen zu können. Im Folgenden wird die Integration von fehlertolerantem MPI in die GCA erläutert.

Eine MPI-Umgebung implementiert Semantiken eines MPI-Standards [116], wie Kommunikationsmodelle, Kommunikationskanäle, Prozesse, Datentypen, et cetera. Diese Semantiken sind dem nativen Betriebssystem, beziehungsweise der Grid-Umgebung unbekannt und können von außerhalb der MPI-Umgebung nicht im Sinne der MPI-Umgebung gesteuert werden. Die GCA sorgt beispielsweise dafür, dass Kommunikationskanäle im Kontext koordinierten Checkpointings geleert werden. *Aufgrund des nicht vorhandenen Wissens um MPI-Elemente, wie MPI-Anwendungs- und MPI-Steuerkanäle, kann die MPI-interne Fehlertoleranzmethodologie zerstört werden.* Dies geschieht beispielsweise, indem die GCA MPI-Steuerkanäle *vor* MPI-Anwendungskanälen leert und blockiert, obwohl funktionsfähige MPI-Steuerkanäle *während* der fehlertoleranten Behandlung der MPI-Anwendungskanäle benötigt werden. Diese Interferenz muss vermieden werden.

Fehlertoleranz für MPI-Anwendungen in einem Grid-Computing-System wird daher erreicht, indem zwar die Sicherung und Wiederherstellung eines MPI-Jobs von der GCA *initiiert*, jedoch von der MPI-Umgebung *ausgeführt* wird. Hierzu ist eine Schnittstelle zwischen GCA und MPI-Umgebung vorgesehen. Anhand dieser Schnittstelle werden Checkpoints und Restarts ausgelöst sowie

- Grid-Checkpointing Metadaten (Abbild-Speicherplatz, -Typ, und -Größe) und
- Monitor-Informationen (Fehlerwahrscheinlichkeit, verfügbarer Speicherplatz)

in beide Richtungen, zwischen MPI-Umgebung und GCA, ausgetauscht. Während die GCA Speichermanagement auf Basis der Metadaten vornimmt, können Monitor-Informationen hilfreich für die MPI-Umgebung sein, um Checkpoint-Aktionen zu planen und zu initiieren.

LAM/MPI [126] führt MPI-Anwendungsprozesse aus und sichert deren Zustände MPI-spezifisch, beispielsweise unter Verwendung spezieller MPI-Steuerungskanäle. In diesem Fall kann die GCA die Job-Hülle sichern und die LAM/MPI-Sicherung initiieren. Ein Job-Hülle bezeichnet die Verwaltungsstrukturen, welche einen Job und dessen Job-Einheiten in einem Grid-Computing-System repräsentieren. Es handelt sich hierbei um Zustände des Grid-Computing-Systems, welche nicht zum Adressraum der nativen Job Einheits-Prozesse gehören. Bei Wiederherstellung wird ein leere Job-Hülle von der GCA rekonstruiert und mit einem MPI-Umgebungsprozess assoziiert, welcher den MPI-Anwendungswiederaufbau initiiert.

3.5.3 GCA-Abläufe im Überblick

Der zeitliche Ablauf einer Job-Eingabe, -Sicherung und -Wiederherstellung wird im Folgenden dargestellt. Aufgrund der Komplexität einzelner Protokolle wird auf das Zusammenspiel involvierter Komponenten fokussiert.

3.5.3.1 Job-Eingabe

Bei der Eingabe eines Jobs in das Grid (engl. job submission) sind folgende Schritte notwendig:

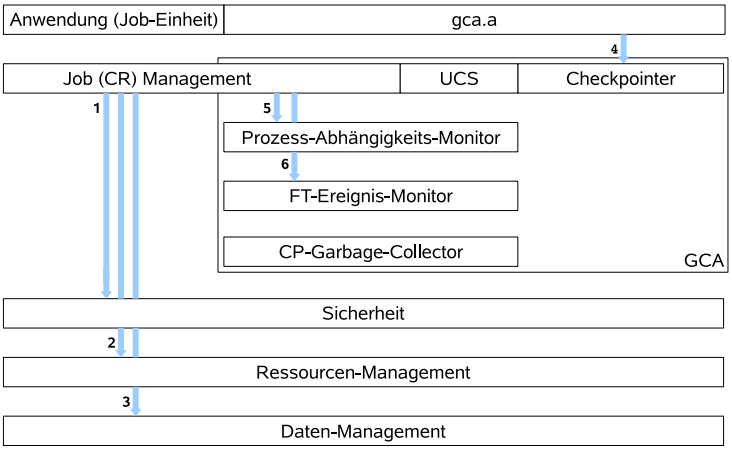

Abbildung 3.8: Involvierte Komponenten bei Job-Eingabe

1. Sicherheitsüberprüfungen garantieren, dass nur authentifizierte und autorisierte Benutzer Jobs erfolgreich starten können.
2. Job-Ressourcen werden unter Berücksichtigung der Kompatibilität von Anwendung und Checkpointern in der VO mithilfe des Ressource-Discovery-Dienstes lokalisiert.
3. Die Ausführungsdatei des ersten Prozesses einer Job-Einheit wird vom Grid-Dateisystem eingelesen und zur Ausführung gebracht.
4. Callbacks werden optional über die Benutzerbibliothek (gca.a) Checkpointer-spezifisch registriert.
5. Prozess-Abhängigkeiten zwischen Job-Einheiten werden nun aufgezeichnet.
6. Der FT-Ereignis-Monitor-Dienst beginnt mit der Überwachung von Ereignissen, die dazu führen, dass ein Job gesichert oder wiederhergestellt werden muss.

3.5.3.2 Systeminitiierter Job-Checkpoint

Initiiert das System in automatisierter Form einen Job-Checkpoint, werden folgende Teilsequenzen durchlaufen:

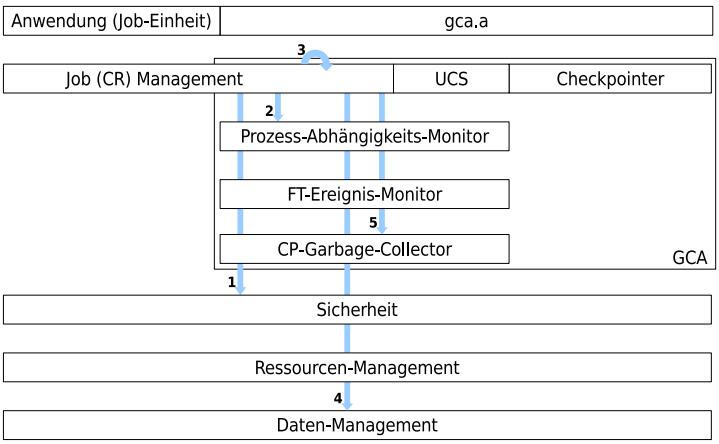

Abbildung 3.9: Involvierte Komponenten bei einer gridinitiierten Sicherung

1. Wird ein Job-Checkpoint ausgelöst, wird auf jedem involvierten Gridknoten überprüft, ob der Initiator authentisiert und authorisiert ist.
2. Abhängig von der eingesetzten Checkpointing-Strategie wird verifiziert, ob Prozessabhängigkeiten zwischen Jobs bestehen, die bei der Job-Sicherung einbezogen werden müssen, beispielsweise bei koordiniertem Checkpointing.
3. Das Job-Checkpoint/Restart-Management initiiert und führt einen Checkpoint aus.
4. Generierte Abbilder werden im Grid-Dateisystem abgelegt. Checkpointer, die keine Sicht darauf haben, werden von der jeweiligen Übersetzungsbibliothek assistiert, indem sie Abbilder vom nativen Dateisystem in das Grid-Dateisystem umkopieren.
5. Überflüssige Abbilder werden vom Garbage-Collection-Dienst ermittelt und entfernt. Diese Aktion wird nebenläufig im Hintergrund, beziehungsweise unmittelbar vor oder nachdem ein Sicherungsvorgang beendet wurde, durchgeführt.

3.5.3.3 Anwendungsinitiierter Job-Checkpoint

Löst die Anwendung einen Job-Checkpoint bei koordiniertem Checkpointing aus, so werden folgende Schritte ausgeführt:

1. Über die GCA-Bibliothek kann die Anwendung Informationen vom FT-Ereignis-Monitor abfragen, um über anstehende Sicherungen entscheiden zu können.

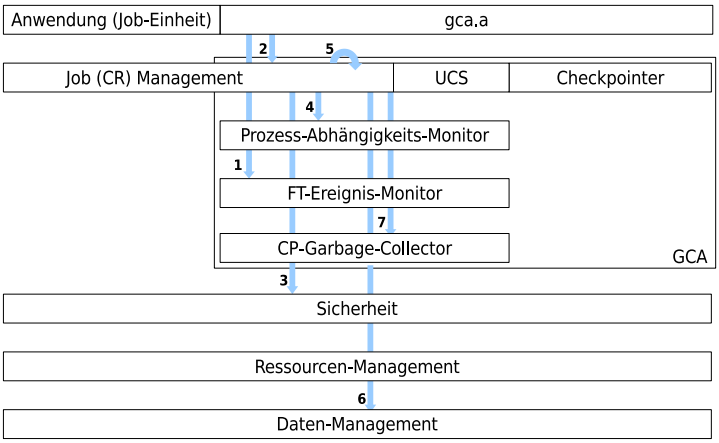

Abbildung 3.10: Involvierte Komponenten beim anwendungsinitiierten Sichern

2. Über die GCA-Bibliothek wird der Job Checkpointer, stellvertretend für die Anwendung, damit beauftragt, einen Checkpoint durchzuführen.
3. Der initiierende Anwendungsbenutzer wird dahingehend überprüft, ob er authentisiert ist und für die Sicherung autorisiert wurde.
4. Der Prozess-Abhängigkeits-Monitor wird aufgefordert, vorhandene Abhängigkeiten des aktuellen Jobs mit anderen zu identifiizieren.
5. Das Job-Checkpoint/Restart-Management führt einen Checkpoint aus.
6. Generierte Abbilder werden im Grid-Dateisystem abgelegt. Checkpointer, die keine Sicht darauf haben, werden von der jeweiligen Übersetzungsbibliothek assistiert, indem sie Abbilder vom nativen Dateisystem in das Grid-Dateisystem umkopieren.
7. Überflüssige Abbilder werden vom Garbage-Collection-Dienst ermittelt und entfernt. Diese Aktion wird nebenläufig im Hintergrund, beziehungsweise unmittelbar vor oder nachdem ein Sicherungsvorgang beendet wurde, durchgeführt.

3.5.3.4 Job-Restart

Um einen Job wiederherzustellen, bedarf es folgender Teilsequenzen:

1. Sicherheitsüberprüfungen garantieren, dass nur authentifizierte und autorisierte Benutzer einen Job-Restart erfolgreich fortsetzen können.
2. Checkpoint Metadaten des Jobs und der zugehörigen Job-Einheit(en) werden vom Grid-Dateisystem eingelesen.
3. Zielknoten werden in Abhängigkeit der Metadaten mithilfe des Ressource Discovery Dienstes ermittelt.

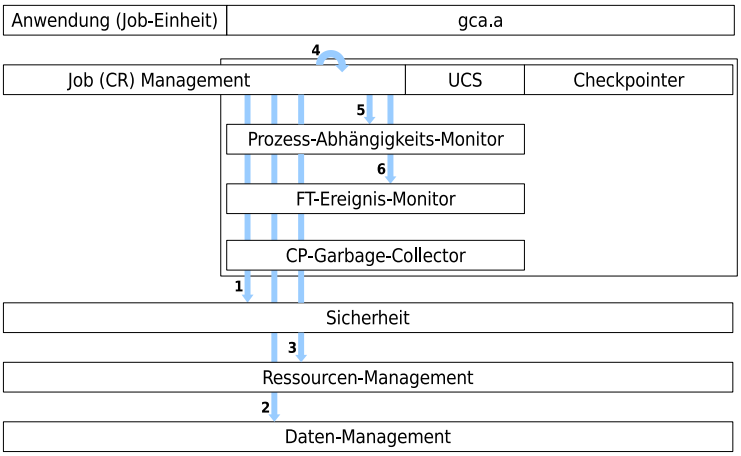

Abbildung 3.11: Involvierte GCA-Komponenten bei Job-Restart

4. Das Job-Checkpoint/Restart-Management führt die Job-Wiederherstellung aus.
5. Nach Job-Wiederherstellung werden Prozessabhängigkeiten wieder aufgezeichnet.
6. Der FT-Ereignis-Monitor fährt damit fort, den wiederhergestellten Job zu überwachen, um auf zukünftige Fehler reagieren zu können.

3.6 Fehlertolerantes Checkpointing

Der Vorgang einer Job-Sicherung oder -Wiederherstellung kann selbst von Fehlern, beziehungsweise Ausfällen gestört werden. Beispielsweise wird ein Grid-Dienst blockiert, wenn Nachrichten verloren gehen, die Voraussetzung für die weitere Dienst-Ausführung sind oder ein Grid-Dienst, infolge eines Laufzeit-Fehlers, abstürzt. Um diese Szenarien zu verhindern, muss Fehlertoleranz auch für (Job-)Fehlertoleranz realisierende GCA-Komponenten selbst bereitgestellt werden.

Der Ausfall eines JCs ist kritisch, da er neben der Koordinator-Funktion im koordinierten Checkpointing Job-Metadaten aufzeichnet, Monitorinformationen auswertet und darauf-aufbauend über Strategiewechsel entscheidet, et cetera. Durch Replikation des *JCs* im Sinne des Virtual Nodes-Ansatzes [38] kann die JC-Verfügbarkeit auch bei Knoten-Ausfällen überwiegend garantiert werden. Voraussetzung hierfür ist, dass der JC als deterministische Zustandsmaschine definiert werden kann, sodass jede JC-Replika dieselbe Ausgabe erzeugt. Dies trifft auf Einzeloperationen eines Checkpointingprotokolls zu, die als Algorithmus festgelegt sind. Beispielsweise werden alle Job-Einheiten angehalten, dann gesichert und anschließend weiter fortgeführt bei koordiniertem Checkpointing. Bei unkoordiniertem Restart werden zunächst Abhängigkeitsanfragen gesendet, dann die Recovery-Line berechnet und anschließend Restart- beziehungsweise Rückrollaufforderungen versendet. Ein

JC arbeitet jedoch nichtdeterministisch im Hinblick auf Entscheidungen einer Checkpointingstrategieänderung. Strategiewechsel ergeben sich für einen JC aufgrund von System- und Anwendungsbeobachtungen, welche sich knotenübergreifend unterscheiden können. Replikation ermöglicht Dezentralisierung dieser bedeutenden GCA-Komponente und reduziert die Gefahr eines Single-Point-of-Failures in hohem Maße.

Zusätzlich zu JC-Ausfällen müssen JEC-Ausfälle behandelt werden. Primär adressiert ein JEC den zugrundeliegenden, lokal installierten Checkpointer. Die Anwendbarkeit aktiver und passiver Replikation muss in diesem Zusammenhang genauer betrachtet werden. Aktive Replikation erfordert in diesem Fall eine *Replikation pro Job-Einheit*, weil eine Job-Einheit dem JEC-zugrundeliegendem Checkpointer lokal vorliegen muss, um gesichert werden zu können. Ein entfernter Checkpointer kann eine nicht-lokal vorliegende Job-Einheit nicht sichern, weil der Checkpointer keinen Zugriff auf die Kernstrukturen des nativen Betriebssystems in der Ferne besitzt. Mit anderen Worten: *JEC-Replikation würde Anwendungs-Replikation erzwingen.* Dies setzt jedoch voraus, dass die Anwendung als deterministische Zustandsmaschine definiert werden kann. Wie in Kapitel 1.2.5 dargestellt, führt letzteres zu großen Anwendungseinschränkungen und bedarf der Installation zusätzlicher Software.

Bei passiver Replizierung erfolgt die Checkpoint-Ausführung pro Job-Einheit nur auf der Primär-Replika, danach wird der Zustand zu den Backup-Repliken übertragen. Das Ergebnis dieses Vorgangs, ein Job-Einheit-Checkpoint, kann jedoch mithilfe des, in der GCA bereits eingesetzten, verteilten Dateisystems erreicht werden, welches Dateien automatisiert repliziert. Daher ist passive Replikation nicht notwendig.

Insgesamt betrachtet eignen sich damit aktive und passive Replikation nicht, um Job-Einheiten parallel zu sichern und wiederherzustellen. Hingegen eignet sich aktive Replikation zur *Aufzeichnung von Job-Einheit-Metadaten*, welches eine Kernaufgabe des JEC's ist. Der JC kann für seine checkpointingprotokollspezifischen Aufgaben repliziert werden, um die Single-Point-of-Failure-Gefahr zu reduzieren.

3.7 Verwandte Arbeiten

Es existieren nur wenige Arbeiten im Grid-Checkpointing-Bereich. Sie werden im Folgenden näher beleuchtet.

3.7.1 CoreGRID

Die Entwicklung einer Next Generation Grid Middleware war der Schwerpunkt des EU-Projekts CoreGRID Network of Excellence (2004-2008), siehe [47]. In der CoreGRID Grid-Checkpointing-Architektur (CG-GCA), [82], löst der Grid Broker Job-Sicherungs- und -Wiederherstellungs-Aktionen, auf Basis von Monitordienst-Informationen, aus und leitet sie über eine Schnittstelle an den darunterliegenden verteilten Grid-Checkpointing-Dienst (GCD). Der GCD verwaltet Checkpoint-Meta-Daten und ermittelt einen oder mehrere Anwendungs-kompatible Übersetzungsdienste (ÜD). Ein ÜD ist Vermittler zwischen GCD und einem Checkpointer. Pro Checkpointer existiert mindestens ein ÜD, der fest mit dem Checkpointer verdrahtet ist. Ein ÜD speichert Aufruf-Semantiken, Checkpointer-Anforderungen und -Funktionalität für den Abgleich von Anwendung mit Checkpointer. Der ÜD agiert als Treiber eines Checkpointers, beziehungsweise Kern-Dienstes (KD), zur Durchführung von Sicherungen und Wiederherstellungen auf unterster Ebene. Ein KD kann Fehlertoleranz auf Kern-, Benutzer- oder Anwendungs-Ebene realisieren.

Virtual Machines werden unter [81] als KDs eingesetzt, dadurch können VM-Abbiklder auch auf anderen, ausgewählten Knoten für die Wiederherstellung verwendet werden. Das unter [32] postulierte Ziel einer skalierbaren Grid-Infrastruktur kann damit erzielt werden. Um die, mit Virtual Machines einhergehenden, Performanz-Einbußen zu reduzieren, jedoch Ressourcen-Virtualisierung, zur Vermeidung von Bezeichner-Konflikten, verwenden zu können, wird der AltixCR Checkpointer, [82], als KD in die GCA aufgenommen, siehe [80].

CoreGRIDs GCA weist strukturelle Gemeinsamkeiten bezüglich der in diesem Kapitel beschriebenen GCA auf. Heterogene Checkpointing-, beziehungsweise Logging-Verfahren werden jedoch nicht integriert, was durch die eingeschränkte GCD-ÜD-Schnittstelle belegt wird. Adaptives Checkpointing-Verhalten kann infolge nicht realisiert werden. Fehlertoleranz-Unterstützung für verteilte Anwendungen, die beispielsweise über Nachrichtenkanäle miteinander kommunizieren, wird ebenfalls nicht dargelegt. Die Integration aktueller, weit verbreiteter leichtgewichtiger Virtualisierungs-Technologien wird auch nicht unterstützt. Obwohl die CG-GCA eine konkrete Architektur darstellt, liegen keine Performanzdaten involvierter CG-GCA-Komponenten vor, wodurch der Stand der Implementierung offen bleibt.

3.7.2 GridCPR

Die unter [18] beschriebenen Anwendungsfälle dienen als Ausgangspunkt zur Beschreibung einer Architektur von Grid Checkpointing- und Wiederherstellungs-Diensten, sowie einer Anwendungs-Schnittstelle zu diesen Diensten, welche die GridCPR Arbeitsgruppe des Global Grid Forums in [14] und [135] darlegt. GridCPR unterscheidet in Checkpoint/Restart der System-Ebene, hier wird ein außerhalb der Anwendung befindliches Werkzeug verwendet, und Checkpoint/Restart der Anwendungs-Ebene. Bei letzterem sind Anwendungen *Verbraucher*, die modifiziert werden müssen, um mithilfe der erwähnten Dienst-Schnittstelle, in Form der GridCPR Bibliothek, sogenannte *Erzeuger*, GridCPR System-Dienste, zu nutzen.

Elementare GridCPR System-Dienste sind das Lesen und Schreiben von Anwendungszuständen, Checkpointdaten-Verwaltung, Fehler- und Ereignis-Benachrichtigung, Job-Verwaltung, Checkpointdaten-Transport, Authentisierung, Autorisierung und Accounting.

Als Schlüsselelement eines Grid Checkpointing Dienstes wird die Wiederherstellung von Jobs auf heterogenen Grid-Ressourcen betrachtet.

Die in dieser Arbeit vorgestellte GCA greift wesentliche Ideen der GridCPR Architektur auf, beispielsweise die Integration indirekt mit der konkreten Sicherung- und Wiederherstellung involvierter Dienste. Weiterhin wird eine Anwendungsschnittstelle zur Architektur dargestellt, welche beispielsweise in der CG-GCA nicht vorkommt. Hingegen bietet dei GCA zusätzlich zu anwendungsinitiiertem Checkpointing jedoch auch anwendungstransparente Fehlertoleranz an. In GridCPR wird kein Bezug zu MPI-Implementierungen hergestellt. Die Bedeutung von Callbacks sowie von adaptivem Checkpointing wird bei GridCPR auch nicht thematisiert. Schliesslich gibt es keine verfügbare GridCPR-Implementierung und daher keine Messwerte.

3.7.3 HPC4U

Im Kontext des HPC4U EU-Projektes (Juni 2004-Mai 2007) wurde eine modulare System-Architektur entwickelt, welche Migration und Fehlertoleranz sequentieller und paralleler Anwendungen in Cluster und Grid-Systemen, aufbauend auf IBM-, Scali- und Dolphin-Produkten, ermöglicht, [128]. Das Checkpointing-Untersystem besteht aus dem IBM MetaCluster HPC, welches insbesondere leichtgewichtige Ressourcen-Virtualisierung mithilfe von Anwendungs-Containern implementiert, um potentielle Ressourcen-Bezeichner-Konflikte (Prozess/Thread-IDs, SYSV IPC und IP-Adressen) bei Prozess-Wiederherstellung auf entfernten Knoten zu vermeiden. Zusätzlich ermöglicht eine spezielle Implementierung des koordinierten Checkpointing Protokolls, MPI-Kanalzustände konsistent zu sichern und wiederherzustellen.

Mit HPC4U wird, im Gegensatz zu GridCPR, eine konkrete Implementierung vorgestellt, welche praktische Herausforderungen Grid Checkpointings (Virtualisierung, Kanalzustands-

Management) adressiert. Durch Fokussierung auf den MetaCluster-Checkpointer und eine MPI-Implementierung ist Unterstützung heterogener Komponenten seitens der Architektur nicht erkennbar.

3.7.4 Weitere relevante Arbeiten

Heterogenes Checkpointing in Kombination mit Checkpoint-Portabilität wird unter [5] dargestellt. OCaml ist eine VM-basierte Programmiersprache, deren Kompiler nativen Maschinencode als auch Bytecode generiert. Bytecode-Checkpoints sind unabhängig von einem zugrundeliegenden Betriebssystem und der Hardware und ermöglichen Abbild-Portabilität, über heterogene Knoten hinweg.

In [42] wird RADIC, eine fehlertolerante Architektur für parallele Cluster-Anwendungen, beschrieben. Ein verteilter Fehlertoleranz-Controller ist fest in eine Implementierung des MPI-1-Standards integriert und realisiert unkoordiniertes Checkpointing, gekoppelt mit pessimistischem, Empfänger-basierten Logging.

Der Ansatz RADICs ist teilweise orthogonal zu dem der GCA, da heterogene Checkpointing-Strategien und Checkpointer nicht unterstützt, jedoch Transparenz, Dezentralisierung, Flexibilität und vor allem Skalierbarkeit im Vordergrund stehen.

Die Sun Grid Engine (SGE) [60] unterstützt Checkpointing auf Anwendungs-, Benutzeradressraum und Kernel-Ebene. Eine SGE Checkpointing-Umgebung ist eine Attribut-Menge, welche eine Checkpointing-Methode detailliert beschreibt. Über einen Konfigurations-Dienst (QMON) kann die Verwendung der stand-alone Condor-Checkpointing-Bibliothek, das Checkpoint-Verzeichnis, Migrations- und Restart-Skripte eingestellt werden. Obwohl verschiedene heterogene Checkpointer eingesetzt werden können, sind Details deren Integration, wie beispielsweise interne Schnittstellen, mögliche Checkpointer-Modifikationen, Kooperation heterogener Checkpointer, etc., nicht ohne Weiteres zu ermitteln. Die in dieser Arbeit vorgestellte GCA integriert heterogene Checkpointer-Pakete und realisiert oberhalb dieser verschiedene Checkpointingprotokolle, letzteres ist Basis für adaptives Checkpointing. Aufgrund entsprechender Schnittstellen und Techniken können Anwendungen Fehlertoleranz selber definieren/implementieren, vorhandene GCA-Dienste explizit nutzen oder in transparenter Weise von der GCA gesichert und wiederhergestellt werden.

3.8 Zusammenfassung (GCA)

Die in diesem Kapitel diskutierte Grid-Checkpointing-Architektur (GCA) realisiert Grid-Fehlertoleranz, aufbauend auf dem Fail-Stop-Modell. Im Zentrum steht die Integration heterogener Checkpointer, um Fehlertoleranz auf unterschiedlichsten Gridknoten zu ermöglichen. Im Gegensatz zu Ansätzen wie des GridCPRs vom Global Grid Forum müssen Anwendungen nicht abgeändert werden, um gesichert und wiederhergestellt zu werden. Dennoch haben Anwendungen die Möglichkeit über eine Schnittstelle auf GCA-Dienste zuzugreifen.

Im Gegensatz zu CoreGRID unterstützt die GCA verteilte, über Nachrichten kommunizierende, Anwendungen. Erstmals erfährt jeder *knotengebundene* Checkpointer eine funktionale Aufwertung, weil er mithilfe der GCA dazu eingesetzt werden kann, *verteilte* Anwendungen zu sichern und wiederherzustellen.

Die GCA integriert zudem leichtgewichtige Virtualisierungsmechanismen und ist in Bezug auf Virtualisierung nicht ausschließlich auf Virtuelle Maschinen angewiesen, wie CoreGRID.

Im Gegensatz zu den meisten bestehenden Checkpointern und existierenden Grid-Checkpointing-Architekturen, wie MetaCluster, CoreGRID, oder RADIC, ermöglicht die GCA *mehr als ein* Checkpointingprotokoll, oberhalb der Checkpointer, auszuführen. Dies ist die Grundlage dafür, adaptives Checkpointing, siehe Kapitel 7 durchzuführen, welches von keiner bestehenden Grid-Checkpointing-Architektur bisher adressiert wird.

Zentrale Architektur-Komponenten stellen der Job-Checkpointer (JC), Job-Einheit-Checkpointer (JEC), die Uniforme Checkpointer-Schnittstelle (UCS), heterogene Checkpointer-Pakete, Fehlermonitore und der Verteilte Kanal-Manager dar. Der JC führt Job-Fehlertoleranz-Management aus. Er delegiert jede Checkpointing- und Wiederherstellungs-Aufforderung pro Job Einheit an den zugehörigen JEC, der auf einem separaten Gridknoten ausgeführt wird. Damit existierende und neue Checkpointer ohne Modifizierung der GCA eingebunden werden können, werden sie vom JEC über die UCS pro Gridknoten in *transparenter* Art und Weise adressiert. Die UCS unterstützt verschiedene Checkpointingprotokolle, ermittelt Abbildabhängigkeiten, registriert Callbacks und ermöglicht, dass Checkpointer miteinander kooperieren bezüglich gemeinsam genutzter Ressourcen. Die UCS wird pro Checkpointer in Form einer Übersetzungsbibliothek implementiert. Jede Übersetzungsbibliothek übersetzt Semantiken der Gridebene auf jene der Gridknotenebene. Des weiteren muss das Kommando des JECs in die Aufrufsemantik des jeweiligen Checkpointers übertragen werden. Um Anwendung und Checkpointer hinsichtlich verwendeter und unterstützter Ressourcen aufeinander abzustimmen, wird das im Grid-Computing verbreitete jsdl-Dateiformat erweitert.

Die GCA integriert existierende MPI-Umgebungen. Eine als Job gekapselte MPI-Anwendung wird von der MPI-Implementierung gesichert und wiederhergestellt. Hierbei beschränkt sich die GCA auf die Sicherung und Rekonstruktion der Job-Hülle, übernimmt

jedoch nicht die Sicherung und Wiederherstellung der MPI-Prozesse und MPI-Kanäle.

4 Prozessgruppen und Container

Aus JEC-Sicht muss ein zugrundeliegender Checkpointer alle zu einer Job-Einheit gehörenden Prozesse sichern und wiederherstellen. Weil auf Grid-Ebene Abstraktionen wie Job und Job-Einheit verwendet werden, Checkpointer hingegen auf Abstraktionen des nativen Betriebssystems wie Prozesse und Threads arbeiten, müssen beide Abstraktionsklassen korrekt aufeinander abgebildet werden. Andernfalls entstehen unvollständige, beziehungsweise zu umfangreiche Sicherungen, siehe Kapitel 4.2.5.3.

Im Zusammenhang mit Prozessen können beim Restart Ressourcenbezeichnerkonflikte auftreten, siehe Kapitel 4.3. Diese können ausgeschlossen werden, wenn der jeweilige Checkpointer Ressourcenvirtualisierung unterstützt.

4.1 Job-Prozess-Assoziation

Um Jobs und deren elementare Prozesse auf Gridebene zu verwalten, müssen *Prozesse und Job-Einheiten einander zugeordnet* werden. Erst wenn ein Prozess dem Job-Management einer Grid-Umgebung bekannt ist, kann es ihm ein UNIX-Signal senden, beispielsweise SIGKILL, um den Prozess zu beenden oder SIGCONT, um die Prozessberechnung nach einer Unterbrechung fortzuführen. Erst wenn das Job-Management erkennt, dass der letzte Job-Einheitsprozess terminiert hat, kann die vormals, mit dem Prozess assoziierte, Job-Einheit beendet werden. Wird ein Checkpointer angewiesen, einen nicht mehr existierenden Prozess zu sichern, kann dies zu Verklemmungen oder zum Systemabsturz führen. Eine wichtige Aufgabe des Job-Managements ist daher, Prozesserzeugungen und -terminierungen zu überwachen.

Die Linux Kern-spezifische Prozess-Implementierung muss bei der Prozessereignissüberwachung beachtet werden.

4.1.1 Prozesse im Linux Kern

Seitentabelle, Befehlszähler und verschiedene Register spezifizieren den Zustand eines Programms [118], beziehungsweise eines Prozesses. Der Hardware-Kontext eines Prozesses entspricht den Daten, die in die Prozessorregister geladen werden müssen, bevor der Prozessor

mit der Prozessausführung beginnt, beziehungsweise fortfährt [25]. Bei einem Kontextwechsel wird der Hardware-Kontext eines Vorgängerprozesses gesichert und mit dem Hardware-Kontext des Nachfolgerprozesses ersetzt.

Prozesse werden mit dem *fork*-Systemaufruf erzeugt, welcher einmal im Elternprozess und einmal im Kindprozess zurückkehrt. Eltern- und Kindprozess können auf Benutzerebene mithilfe von PID (process id) und PPID (parent process id) identifiziert werden. Mithilfe des Copy-on-Write (COW) Verfahrens können Eltern- und Kindprozess bis zum ersten Schreibzugriff einer Partei von denselben Kacheln lesen, sodass eine unmittelbare vollständige Kopieroperation des Elternprozesses in den Kindprozess nicht notwendig ist.

Ein Prozess besteht mindestens aus einem Thread. Dabei besitzt jeder Thread seinen eigenen Stack, lokale Variablen und Programmzähler. Mehrere Prozessthreads nutzen alle Prozessressourcen gemeinsam, wie offene Deskriptoren und globale Variablen. Auf einem Prozessorkern kann zu einem Zeitpunkt genau ein Prozessthread ausgeführt werden.

Multithreading wird verwendet, um Nebenläufigkeit zu realisieren, anstelle mehrere Prozesse erzeugen zu müssen. Letzteres erlaubt einem Prozess mehrere Ein-/Ausgabe-Anfragen überlappend auszuführen, ohne dass eine von ihnen die *gesamte* Programmausführung blockiert. Mehrere Threads eines Prozesses können gleichzeitig auf mehreren Prozessorkernen ausgeführt werden, wenn die zugrundeliegende Architektur dies unterstützt.

Threads, welche im Benutzeradressraum sichtbar sind, werden auf Threads im Kern abgebildet, dabei existieren drei Modelle (M:1, 1:1, M:N). Das M:1 Modell realisiert *Benutzeradressraum-Ebenen-Threads* [59]. Der Linux-Kernel sieht genau einen Prozessthread, obwohl dieser aus M Threads auf Benutzeradressraumebene besteht, siehe [34]. Die Threadimplementierung befindet sich hierbei im Benutzeradressraum in Form von Thread-Bibliotheken, wie beispielsweise *glibc*. Nach diesem Modell blockieren in der Regel alle Prozessthreads, falls *ein* Thread einen blockierenden Systemaufruf absetzt. Um dies zu verhindern, muss der Programmierer *explizit* nicht-blockierende Techniken einsetzen, beispielsweise den *select*-Aufruf für asynchrone Ein-/Ausgabe. Darüber hinaus kann ein Prozess mit diesem Thread-Modell die Ressourcen eines Mehrkernrechners nicht vollständig ausnutzen.

Die Thread-Modelle 1:1 und M:N realisieren *Kernel-Threads* [59] und sind damit effizienter als das M:1 Modell. Linux implementiert das 1:1 Modell anhand sogenannter *leichtgewichtiger Prozesse*. Da jeder (Kernel-)Thread von einem einzigen Scheduler angewiesen wird, werden potentielle Blockaden in Verbindung mit dem M:1 Modell transparent vom Kern behandelt. Signal-Management bleibt ebenfalls in der Hand des Kerns. Einen Nachteil dieses Ansatzes stellen die häufigen Kontextwechsel dar, welche durch den Kernel-Scheduler verursacht werden. Bibliotheken, die leichtgewichtige Prozesse verwenden, sind LinuxThreads, Native POSIX Thread Library (NPTL) und Next Generation Posix Threading Package (NGPT) von IBM [25].

Mit New Generation POSIX Threads (NGPT) wird die Implementierung des M:N Modells realisiert, welches die Anzahl der Kontextwechsel zwischen mehreren Kernel-Threads minimiert. Problematisch ist jedoch, dass zwei miteinander kooperierende Scheduler (auf

Kern- und Benutzeradressraum-Ebene) zwingend vorhanden sein müssen sowie ein kompliziertes Signal-Management [34]. Weil das 1:1 Modell weitgehend verwendet wird, wurde die Implementierung 2003 eingestellt.

Im Linux 2.6.x Kern besitzt jeder leichtgewichtige Prozess einen *struct task_struct* Deskriptor mit einer eindeutigen Kennung, welche im *pid*-Feld gespeichert wird. Alle Threads eines Prozesses werden zu einer *Threadgruppe* zusammengefasst, deren Kennung im *tgid* Feld des Deskriptors gespeichert wird. Ein Threadgruppenanführer ist der erste leichtgewichtige Prozess einer Threadgruppe. Seine PID definiert die Threadgruppenkennung. Threads teilen eine Threadgruppe, falls deren *tgid* Wert der PID des Threadgruppenanführers entspricht. Dieser Mechanismus ermöglicht, dass Signale an alle Prozessthreads gesendet werden können, indem nur *eine einzige* Kennung angegeben wird.

4.1.2 Prozessüberwachung mit dem Linux Process Event Connector

Für die Abbildung von Prozessen auf einen Job, beziehungsweise eine Job-Einheit, muss jede Prozesserzeugung und -terminierung *anwendungstransparent* vom Job-Management der Grid-Umgebung überwacht werden.

Der initiale Prozess A einer Job-Einheit wird vom Job-Management mit dem Aufruf *fork* erzeugt. Ist der Aufruf erfolgreich, kehrt der Elternprozess E zurück und übergibt die PID des Kindprozesses A unmittelbar zurück. Hierdurch kann der Job-Einheit die erste PID direkt zugeordnet werden. Die Herausforderung besteht jedoch darin, *die sich anschließenden Prozessereignisse* korrekt zu ermitteln. Führt nachfolgend Kindprozess A *fork* durch, kehrt dieser Aufruf nicht zu einem Prozess des Job-Managements aber der Job-Einheit zurück. Ohne weitere Hilfen bleibt diese Prozesserzeugung, innerhalb einer Job-Einheit, dem Job-Management verborgen.

Eine Prozesszuordnung zu einem Job kann mithilfe des bereits in Kapitel 3.3.1 skizzierten Linux Process Event Connectors (PEC) realisiert werden. Hierbei registriert sich der Job-Einheit-Manager, siehe Kapitel 1.4.7, als Überwachungsprozess beim PEC im Linux Kern für *fork*- und *exit*-Aufrufe, die bereits bekannte Job-Einheitsprozesse initiiert haben.

Da der PEC jeden *copy_process*-Aufruf im Kern abfängt, welcher auch bei Erzeugung eines leichtgewichtigen Prozesses aufgerufen wird[1], müssen Threadgruppenanführer von anderen Threads des Adressraums, voneinander unterschieden werden. Eine Prozesserzeugungs-Nachricht darf daher nur dann an den Überwachungsprozess gesendet werden, *wenn der Threadgruppenführer copy_process* aufruft, das heißt, wenn das *tgid* Feld das *task_struct* Dekriptors dem *pid* Feld entspricht. Andernfalls wird eine Multithreadanwendung durch das Job-Management als Multiprozessanwendung interpretiert, was fehlerhaft ist. Bei der Integration von Checkpointern in die GCA müssen alle Prozesse und deren Threads bekannt sein, um die Aufteilung der nativen Checkpointsequenz zu realisieren. Eine Übersetzungsbibliothek kommuniziert mit der Checkpointer-Bibliothek über Nachrichtenwar-

[1] Die *do_copy* Kernfunktion wird neben *fork* auch von *clone* aufgerufen, um Kernel-Threads zu erzeugen.

teschlangen. Eine Verklemmung erfolgt, wenn die Übersetzungsbibliothek auf Nachrichten bezüglich eines Prozesses X wartet, der, infolge einer fehlerhaften Prozess-Thread-Interpretation auf Gridebene, gar nicht existiert.

4.2 Prozessgruppen und Container

Heterogene Checkpointer verwenden unterschiedliche Prozessgruppentypen und Container, anstelle einer Prozessliste, um eine Anwendung zu sichern oder wiederherzustellen. Anhand eines *Prozessgruppenbezeichner* ermittelt der Kern später zum Checkpoint-Zeitpunkt alle zur Prozessgruppe gehörenden Prozesse. Darüber hinaus existieren diverse Containertechnologien, die Ressourcenbezeichnerkonflikte, siehe Kapitel 4.3, verhindern, die jedoch nicht gleichzeitig von allen Checkpointern unterstützt werden.

Damit eine Job-Einheit zum Checkpointzeitpunkt korrekt auf eine Prozessgruppe und einen Container abgebildet werden kann, sind Checkpointer-spezifische Vorbereitungsmaßnahmen, siehe Kapitel 4.2.5.2, notwendig. Erst hierdurch wird die Grundlage gelegt, nur die zur Anwendung gehörenden Prozesse zu sichern und wiederherzustellen.

Im Folgenden werden relevante Prozessgruppen und Containertechnologien beschrieben.

4.2.1 Prozessbaum

Prozesse können anhand von Eltern-Kind-Beziehungen in Prozessbäumen angeordnet werden. Im Normalfall erkundigt sich der *erzeugende Elternprozess* über die Terminierungszustand seiner Kinder, um die Resultate zu ermitteln. Letzteres kann mithilfe des *wait*-Aufrufs realisiert werden, nach dessen Rückkehr der entsprechende Prozessdeskriptor vom Elternprozess freigegeben wird. Bis dahin befindet sich ein terminierter Prozess in einem gesonderten Zustand, dem sogennanten Zombie-Zustand.

Terminiert ein Elternprozess *vor* seinem Kindprozess, wird letzterer zum Waisen (engl. orphan process). Das heißt insbesondere, dass kein Elternprozess existiert, welcher Verwaltungsdaten des Kindprozesses nach dessen Terminierung aus dem System entfernt. Dies führt zu einem Speicherleck, da diese Daten nicht mehr benötigt werden. In einem UNIX-basierten Betriebssystem übernimmt an dieser Stelle der Systemprozess *init*, welcher die PID 1 besitzt und während der Systeminitialisierung erzeugt wird, Aufräumarbeiten.

Werden Prozesswaisen zu Kindern des *init*-Prozesses, überwacht er die Ausführung all seiner Kinder, indem er in regelmäßigen Abständen den *wait4* Systemaufruf ausführt, um Orphan-Ressourcen freizugeben. Konsequenterweise kann der Wurzelprozess eines Prozessbaumes nicht dazu verwendet werden, um zu jedem Zeitpunkt all seine direkten und indirekten Kindprozesse eindeutig zu referenzieren, insbesondere nachdem der Wurzelprozess terminiert hat. Der *init*-Prozess identifiziert keine *Prozessteilbäume*, die vormals zu einem Prozessbaum gehört haben. Der *init*-Prozess kann dadurch Vaterprozess *mehrerer,*

separater Prozessteilbäume sein.

In Abbildung 4.1 wird demonstriert, wie Prozess E zum Waisen und anschließend *init* zugeordnet wird. Der Verlust des erzeugenden Eternprozess und die Zuordnung des *init*-Prozesses als neuen Eternprozess wird als *re-parenting* bezeichnet.

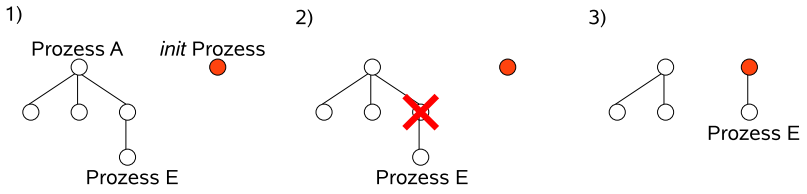

Abbildung 4.1: *init*-Prozess wird neuer Eltern-Prozess nach Prozessterminierung

4.2.2 Prozessgruppen und Sessiongruppen in UNIX

Lesen zwei Prozesse vom selben Terminal zur selben Zeit, besteht keine Garantie, dass jeder Prozess genau die Zeichen erhält, die für ihn bestimmt sind. Die BSD-Gruppe gruppierte daher Prozesse und konnte damit zwischen Vordergrund- und Hintergrundprozessgruppen bei UNIX-Shells unterscheiden. Wird auf der Kommandozeile Folgendes eingegeben:

```
$ ls | sort | more
```

wird beispielsweise eine neue Prozessgruppe erzeugt, welche die Prozesse *ls*, *sort* und *more* zu einer logischen Einheit, einer sogenannten *UNIX Prozessgruppe*, zusammenfasst. Nur die Vordergrund-Prozessgruppe erhält lesenden und schreibenden Zugriff auf das Terminal. Eine Hintergrund-Prozessgruppe wird mit den Signalen SIGTTIN oder SIGTOU bei einem Leseversuch blockiert.

Prozessgruppen sind eng mit Signal-Management verbunden. Wird ein Signal an eine Prozessgruppe gesendet, wird es an alle Prozesse der Gruppe ausgeliefert[2]. Hierdurch lässt sich eine Job-Kontrolle realisieren. Unter POSIX kann ein Signal an eine UNIX Prozessgruppe mit folgender Funktionssignatur gesendet werden:

```
int kill(int -process_group, int signal_number);
```

Hierbei ist zu beachten, dass die Prozessgruppen-ID negativ ist.

Initial wird ein neu erzeugter Prozess der UNIX Prozessgruppe seines Elternprozesses zugeordnet. Jeder Prozessdeskriptor enthält eine UNIX Prozessgruppen-ID (PGID). Jede

[2]POSIX, BSD oder System V Implementierungen unterscheiden sich teilweise im Bezug auf Signal-Management, Prozessgruppen und Terminalverwaltung [57].

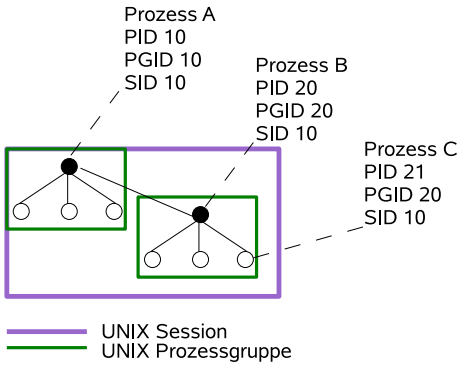

Abbildung 4.2: UNIX Session- und Prozessgruppe

UNIX Prozessgruppe besitzt einen Anführer, dessen PID der PGID entspricht, siehe Abbildung 4.2. Neben der UNIX Prozessgruppe existiert eine weitere, höhergeordnete Prozessgruppe. Meldet sich ein Benutzer am System an, wird eine Login-Session erzeugt. Der Session-initiierende Prozess, sowie all seine Nachkommen, werden initial in einer sogenannten *UNIX-Session* gruppiert. Eine UNIX-Sessiongruppe wird anhand einer Session-ID (SID) referenziert und besteht mindestens aus *einer* UNIX-Prozessgruppe, welche sich immer im Vordergrund befindet und damit Zugriff auf das zur Session gehörende Terminal besitzt. Versucht ein Hintergrundprozess auf das Terminal zuzugreifen, erhält er das SIGTTIN oder SIGTTOUT Signal. Der Session-Anführer ist gleichzeitig Anführer der initialen UNIX-Prozessgruppe. Abbildung 4.2 verdeutlicht das Verhältnis beider Gruppierungstechniken unter Angabe von Prozessbeziehungen und relevanten ID's.

Die Besonderheit von UNIX-Sessions und UNIX Prozessgruppen, im Vergleich zu einfachen Prozessbäumen, besteht darin, dass ein Kindprozess, insbesondere nachdem dessen Elternprozess terminiert hat, seiner Prozessgruppe zugehörig bleibt und über die Prozessgruppenkennung weiterhin referenziert werden kann, siehe Abbildung 4.3.

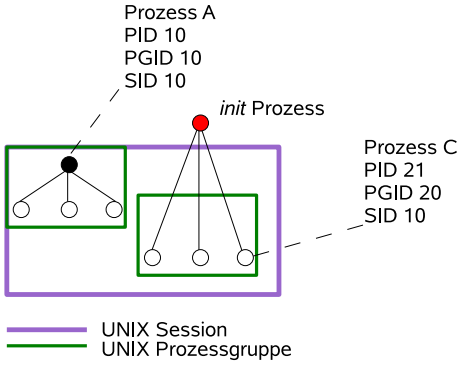

Abbildung 4.3: Session- und Prozessgruppenzugehörigkeit nach Verlust des Elternprozesses

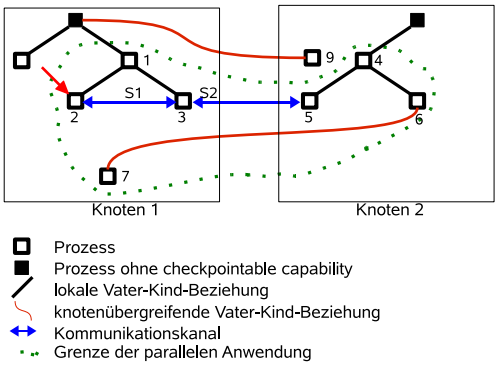

Abbildung 4.4: Abgrenzung einer LinuxSSI-Anwendung

4.2.3 LinuxSSI Prozessgruppen

In einem LinuxSSI, beziehungsweise Kerrighed, Cluster können ein oder mehrere Prozesse einer parallelen Anwendung zwischen Knoten migriert werden. LinuxSSI-spezifische Datenstrukturen (Kerrighed Distributed Data Management [94]) garantieren, dass Eltern-Kind-Beziehungen, insbesondere im Fall von Prozessmigration, beibehalten werden. Zusätzlich gewährleisten Kerrigheds Dynamic Streams [58], dass zwei, über einen Kommunikationskanal miteinander verbundene, Prozesse auch nach einer Prozessmigration weiterhin verbunden bleiben.

LinuxSSI spezifiziert einen Prozessgruppentyp, der sich, im Gegensatz zu Prozessbäumen, UNIX Session- und UNIX Prozessgruppen nativer UNIX-basierter Betriebssysteme, dynamisch über mehrere Knoten erstrecken kann. Der LinuxSSI-Checkpointer verwendet die Kennung einer LinuxSSI-Anwendung (LinuxSSI-AppID), um eine Anwendung im LinuxSSI-Cluster zu sichern und wiederherzustellen.

Die in Abbildung 4.4 dargestellten Prozesse innerhalb der grün-gepunkteten Umrandung gehören zu einer LinuxSSI-Anwendung. Ausgehend von dem mit einem roten Pfeil gekennzeichneten Prozess werden alle zur LinuxSSI-Anwendung gehörenden Prozesse ermittelt, indem Eltern-Kind-Beziehungen und die Enden von Kommunikationsströmen zurückverfolgt werden. Damit bei der Rückverfolgung nur die zur Anwendung gehörenden Prozesse ermittelt werden, wird bei jedem Prozess der Status eines LinuxSSI-spezifischen Prozessattributs, das sogenannte CHECKPOINTABLE capability [51], abgefragt. Dieses Attribut muss *beim Anwendungsstart* explizit vom Benutzer gesetzt werden, damit es die Anwendung von anderen Prozessen des Systems abgrenzt.

4.2.4 Container im Kontext von Checkpointing

Neben den beschriebenen Prozessgruppen existiert eine weitere Gruppierungstechnik. Sogenannte Container werden eingesetzt, um Ressourcen verschiedener Anwendungen voneinander abzugrenzen. Container sind im Fehlertoleranzkontext sehr wichtig, weil mit ihrer Hilfe potentielle Ressourcenkonflikte, siehe Kapitel 4.3, beim Restart vermieden werden können. Verschiedene Checkpointer verwenden unterschiedliche Container, um zu sichernde und wiederherzustellende Prozesse zu adressieren. Allgemein existieren *System- und Anwendungscontainer*, die im Folgenden näher betrachtet werden.

4.2.4.1 OpenVZ Container

OpenVZ [139] basiert auf Linux, wurde aber aufgrund der umfangreichen, notwendigen Modifizierung des Linux' Quelltextes bis dato nicht in Mainline-Linux integriert. OpenVZ implementiert einen Systemcontainer, der aus einem Template besteht, welches Systemprogramme, Bibliotheken und Skripte enthält, um eine eigenständige Systemumgebung innerhalb eines Containers aufzubauen. *OpenVZ verbindet Container- mit Checkpoint/Restart-Funktionalität*, das heißt eine Containerumgebung kann pausiert, vollständig gesichert und auf demselben oder einem anderen Knoten wiederhergestellt werden, ohne dass Ressourcenkonflikte auftreten.

Die Abbildung einer Job-Einheit auf einen OpenVZ-Container wird erschwert, weil OpenVZ-Patches, Quelltext-Anpassungen des Linux Kerns, die Process Event Connector Funktionalität beeinträchtigen. Mit PEC können bei OpenVZ keine Prozessereignisse an das Job-Management weitergeleitet werden. Da jedoch eine Job-Einheit nur solange aktiv ist, solange mindestens ein Container-Prozess existiert, müssen Prozessereignisse mithilfe anderer Techniken an das Job-Management weitergeleitet werden.

Anstelle einer Prozessgruppen-ID wird die Container-ID benötigt, damit der JEC den zugrundeliegenden OpenVZ Checkpointer mit der Container-Sicherung und -Wiederherstellung beauftragen kann. Die korrekte Abbildung von Job-Einheit auf Container muss deshalb sichergestellt werden.

4.2.4.2 Cgroup Container

Das *cgroup*-Framework stellt einen Linux-nativen Anwendungscontainer dar, der für viele Linux-basierte Checkpointer von Interesse ist. Cgroups ordnen Prozesse und deren zukünftige Kinder in hierarchischen Gruppen, die jeweils mit einem spezifischen Verhalten ausgestattet werden können [70]. Cgroups allein betrachtet werden nur für die Jobüberwachung verwendet. Im Zusammenhang mit sogenannten Subsystemen, siehe Kapitel 4.3.2.1, kann jedoch Ressourcenbuchführung und -virtualisierung, et cetera realisiert werden, siehe Kapitel 4.3.2.

Im Gegensatz zu OpenVZ ist eine *cgroup* nicht gleichzeitig mit einer Sicherungs- beziehungsweise Wiederherstellungsfunktionalität ausgerüstet. Dennoch lässt sich, die für koordiniertes Checkpointing notwendige, Prozesssynchronisierung mithilfe des *cgroup Freezers*, früher als *Container Freezer* bezeichnet [70], realisieren. Der *cgroup Freezer* kann mithilfe der sogenannten Linux Container Tools (LXC) vom Benutzeradressraum aus gesteuert werden, [72].

Während *cgroup*-Entwickler zwischen Virtual Private Server (VPS) und Checkpoint/Restart (CR) unterscheiden, wird derzeit ein Linux-nativer Checkpointer entwickelt, der eng mit *cgroups* verknüpft ist, siehe Quelltext-Datei 'checkpoint_restart.c' in [84]. Derzeit ist jedoch unklar, welche ID bei diesem Checkpointer verwendet werden muss (*cgroupID*, PID, PGID oder SID), um eine Anwendung, *von der Übersetzungsbibliothek aus*, zu sichern, beziehungsweise wiederherzustellen[3]. Der *cgroup Freezer* verwendet als ID beispielsweise den Dateisystempfad einer *cgroup*, unter der sie im System eingebunden ist [71]. Mit welcher ID der Linux-native Checkpointer angestoßen werden kann, um von *außerhalb der Anwendung* einen Checkpoint zu initiieren, wird sich erst in Zukunft ergeben.

In Bezug auf die Abbildung einer Job-Einheit auf eine Prozessgruppe ist jedoch wesentlich, dass ein Checkpointer-spezifischer Container, in diesem Fall eine *cgroup*, zunächst erzeugt werden muss, bevor die Job-Einheit ausgeführt werden kann.

4.2.5 Synthese

Tabelle 4.1 stellt existierende, heterogene Prozessgruppierungstechniken dar. Die Auswahl der optimalen Technik hängt primär davon ab, welcher Checkpointer lokal auf einem Gridknoten vorhanden ist und demnach verwendet werden soll. Hierbei sind grundsätzlich jene Checkpointer zu bevorzugen, die Ressourcenvirtualisierung realisieren, um potentielle Ressourcenkonflikte beim Restart zu vermeiden, siehe Kapitel 4.3. Deshalb kommen insbesondere Container-unterstützende Checkpointer in Betracht.

Wurde eine Auswahl getroffen, *muss abgesichert werden, dass eine Prozessgruppe oder ein Container korrekt auf eine Job-Einheit, während der gesamten Anwendugsausführungszeit, abgebildet wird.* Das heißt insbesondere, dass:

- das Job-Management über alle Prozessereignisse informiert werden muss, um Prozesslisten zu verwalten und Job-Einheiten gegebenenfalls terminieren zu können.
- Checkpointer-spezifische Vorbereitungsmaßnahmen während der Job-Einreichung und der Job-Wiederherstellung vorzunehmen sind, damit ein Checkpointer nur die Prozesse sichert und wiederherstellt, die tatsächlich zur Job-Einheit gehören.

[3] Derzeit ermöglicht der Linux-native Checkpointer *anwendungsinitiiertes Checkpointing*, wobei anstelle einer cgroupID die task_struct Kernstruktur des initiierenden Prozesses übergeben wird. Weil jede task_struct einen Verweis auf die *cgroup* besitzt, in welcher der zugehörige Prozess angeordnet ist, kann die assoziierte *cgroup* kernintern ermittelt werden.

Checkpointer	Prozessgruppe/Container
BLCR	Prozessbaum, UNIX-Prozessgruppe. und UNIX-Session.
LinuxSSI	LinuxSSI-Anwendung[a]
Linux-nativ	Cgroups-strukturierte Prozessgruppen.
OpenVZ	OpenVZ Container.
Metacluster	Metacluster Container[b].
Zap	Pods.

[a] Aktuelle Entwicklungsarbeiten in LinuxSSI deuten darauf hin, dass jede LinuxSSI-Anwendung in einer cgroup gekappselt, jedoch der existierende Prozessgruppentyp vom LinuxSSI-Checkpointer weiterhin zur Referenzierung alle Anwendungsprozesse verwendet wird.

[b] IBM Entwickler verwenden im proprietären Metacluster einen eigenen Container, der zur Entwicklung von cgroups im Linux Kern beigetragen hat.

Tabelle 4.1: Von Checkpointern unterstützte Prozessgruppen und Container

4.2.5.1 Weiterleitung von Prozessereignissen

Prozessereignisse können in anwendungstransparenter Art und Weise mithilfe des Linux Process Event Connectors an einen Überwachungsprozess des Job-Managements übertragen werden.

Da Netlink-Nachrichten des PEC's nicht aus einem OpenVZ-Container an das Job-Management gesendet werden können, wurde ein außerhalb eines OpenVZ-Containers positionierter Server entwickelt, der Prozessereignisse aus dem Containerdateisystem herausliest und an das Job-Management weiterleitet.

4.2.5.2 Checkpointer-spezifische Vorbereitungsmaßnahmen und Übersetzungen

In Tabelle 4.1 werden verschiedene Prozessgruppen- oder Container-basierte Checkpointer aufgelistet. Eine Job-Einheit muss explizit entweder in der mit dem Checkpointer assoziierten Prozessgruppe oder im entsprechenden Container gekapselt werden. Vor einem Checkpoint muss eine Job-Einheit auf einer Prozessgruppen-ID oder eine ContainerID übersetzt werden, welche der zugrundeliegende Checkpointer verwendet.

Wie unter 4.2.1 dargelegt, genügt der Wurzelprozess nicht, um zu jedem Zeitpunkt nur zur Anwendung gehörende Prozesse zu adressieren. Damit BLCR eine Job-Einheit korrekt beim Checkpinting/Restart adressiert, muss *jede Job-Einheit explizit in einer UNIX Prozessgruppe oder UNIX Session gekapselt werden.* Bei der Job-Einreichung legt daher der erste Job-Einheitsprozess eine Prozessgruppe mit *setpgrp*, beziehungsweise eine Session mit *setsid* an, bevor exec* aufgerufen wird.

Zum Checkpointzeitpunkt muss die Checkpointer-spezifische Übersetzungsbibliothek die

jeweilige Prozessgruppen-ID, auf Basis der vom Job-Management bereitgestellten Prozessliste der Job-Einheit, ermitteln. Nach der Übersetzung übergibt die Übersetzungsbibliothek die jeweilige Prozessgruppen-ID und deren Typ an den zugrundeliegenden Checkpointer.

Prozessgruppen-ID und -typ sind Teil der Job-Einheit-Metadaten, siehe Kapitel 3.2.2.2, weil sie Teil des Abbildnamens sind und somit wesentliche Informationen für einen erfolgreichen Neustart darstellen.

Bei LinuxSSI hingegen muss bei der Job-Einreichung beim ersten Prozess das sogenannte CHECKPOINTABLE capability gesetzt werden, damit nur zur LinuxSSI-Anwendung gehörende Prozesse in die Sicherung einbezogen werden. Die LinuxSSI-Übersetzungsbibliothek ermittelt unmittelbar vor dem Checkpointingzeitpunkt die LinuxSSI-Anwendungs-ID, auf Basis der Prozessliste, die pro Job-Einheit vom Job-Management bereitgestellt wird und übergibt sie dem Checkpointer.

Werden Container-Checkpointer eingesetzt, muss der jeweilige Container, als vorbereitende Maßnahme, explizit bei der Job-Einreichung, pro Job-Einheit, angelegt werden. Im Fall von *cgroups* wird im *cgroup*-Dateisystem ein entsprechender Eintrag erzeugt, Subsysteme, siehe Kapitel 4.3.2.1, werden mit der *cgroup* assoziiert.

Wird OpenVZ in die GCA integriert, muss ein OpenVZ-Container pro Job-Einheit angelegt, initialisiert und gestartet werden. Insbesondere muss der Server gestartet werden, welcher Prozessereignisse an das Job-Management weiterleitet.

4.2.5.3 Fallbeispiele fehlerhafter Abbildungen

Werden Checkpointer-spezifische, vorbereitende Maßnahmen und Übersetzungen, siehe Kapitel 4.2.5.2, nicht im Einklang miteinander ausgeführt, können fehlerhafte Job-Einheit-Prozessgruppenabbildungen entstehen. Hierbei treten zwei charakteristische Fälle auf, *entweder werden zu viele oder zu wenige Prozesse gesichert, beziehungsweise wiederhergestellt.*

Abbildung 4.5 veranschaulicht den Fall, wenn zu viele Prozesse gesichert werden. Wäh-

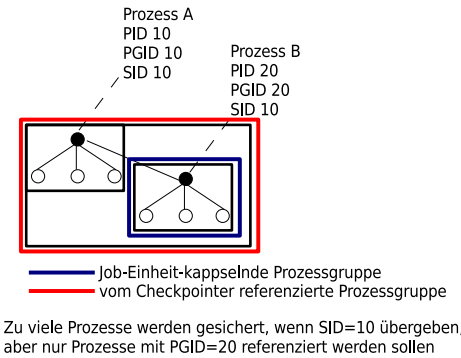

Abbildung 4.5: Zu viele Prozesse werden gesichert.

rend Prozess A und dessen linksseitige Kindprozesse Dienste des Grid-Computing-Systems repräsentieren, entsprechen Prozess B und dessen Kindprozesse einer Job-Einheit. Alle Prozesse befinden sich in derselben UNIX Session. Wird dem Checkpointer fälschlicherweise die UNIX SessionID, in Abbildung 4.5 SID=10, übergeben, weil beispielsweise Prozessereignisse fehlerhaft zugeordnet werden, werden bei der Sicherung auch die Prozesse des Grid-Computing-Systems, Prozesse mit Gruppen-ID=10, einbezogen, was unter anderem zu deren Pausierung und Sicherung führt. Das Grid-Computing-System ist demnach temporär nicht aktiv, zusätzlich wird mehr Speicherplatz verbraucht als notwendig. Erst wenn

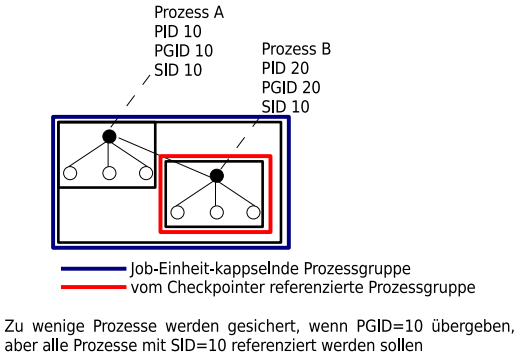

Abbildung 4.6: Zu wenige Prozesse werden gesichert.

die Job-Einheitsprozesse in einer UNIX Prozessgruppe gekapselt sind, um sie von den Prozessen des Grid-Computing-Systems logisch zu separieren, und andererseits deren UNIX Prozessgruppen-ID an den Checkpointer übergeben wird, ist die Abbildung korrekt.

In Abbildung 4.6 gehören alle Prozesse zu einer Job-Einheit und befinden sich in einer separaten UNIX Session. Zur Laufzeit können jedoch mehrere UNIX Prozessgruppen innerhalb der Session entstehen. Wird infolge einer fehlerhaften Abbildung nur die ID der neuen UNIX Prozessgruppe um Prozess B an den Checkpointer übergeben, werden zu wenige Prozesse gesichert, wodurch eine Inkonsistenz entsteht. Auch hier müssen die vorbereitenden Maßnahmen (Erzeugung einer UNIX Session) mit der Übersetzung kooperieren, damit die Session-ID übergeben wird.

Einfache Prozessbäume können ebenfalls zur fehlerhaften Sicherung führen, wenn der Wurzelprozesses oder ein Prozess zwischen Wurzel- und Blattprozess terminiert. In Abbildung 4.1, siehe Kapitel 4.2.1, wird Prozess E, nach Verlust seines erzeugenden Elternprozesses, dem *init*-Prozess zugewiesen. Prozess E ist vom ehemaligen Wurzelprozess A nicht mehr erreichbar, wenn die aktuell gültigen Eltern-Kind-Beziehungen verwendet werden. In diesem Fall wird Prozess E nicht mit in die Sicherung einbezogen, wenn dem Checkpointer die Wurzel dieses Prozessbaumes übergeben wird.

Bei allen drei Fällen ist insbesondere darauf zu achten, welche Prozessgruppensemantik von einem zugrundeliegenden Checkpointer unterstützt wird. Kann ein Checkpointer beispielsweise nur UNIX Prozessgruppen sichern, muss jede Job-Einheit in einer UNIX Session

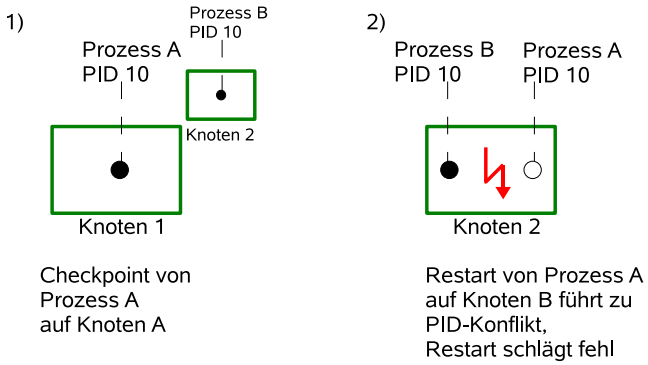

Abbildung 4.7: Ressourcenkonflikt beim Restart

gekapselt werden.

Da zur Laufzeit UNIX Prozessgruppen und Session auch dynamisch erzeugt werden können, kann die Abbildung auch auf Anwendungsebene gestört werden. Dies trifft insbesondere zu, wenn mehrere UNIX Sessions pro Job-Einheit erzeugt werden. An dieser Stelle kann die korrekte Abbildung nur dann garantiert werden, wenn Session-erzeugende Aufrufe unterbunden werden. Werden hingegen weitere UNIX Prozessgruppen innerhalb einer Session erzeugt, ist das unproblematisch, weil die UNIX Session die Prozessgruppe höherer Ordnung ist und mit ihrer ID alle UNIX Prozessgruppen implizit bei der Sicherung und Wiederherstellung erfasst werden.

4.3 Vermeidung von Ressourcenkonflikten

Ressourcenkonflikte treten auf, wenn eine Ressource von einem Prozess A benötigt wird, die bereits bei Prozess B in Verwendung ist. Abbildung 4.7 veranschaulicht diesen Aspekt im Fehlertoleranzkontext. Prozess A besitzt die PID 10 und wird auf dem Gridknoten 1 gesichert. Im Zuge einer Migration soll Prozess A auf Gridknoten 2 wiederhergestellt werden. Beim Wiederaufbau kann Prozess A nicht mit der PID 10 assoziiert werden, da auf Gridknoten 2 bereits ein Prozess B mit PID 10 existiert. Der Wiederaufbau scheitert aufgrund dieses Ressourcenkonfliktes. Diese Konflikte können im Bezug auf Bezeichner von Prozessen, Semaphoren, gemeinsam genutzten Speichersegmenten und Nachrichtenwarteschlangen auftreten.

4.3.1 Schwergewichtige Virtualisierung in Virtuellen Maschinen

Ressourcenvirtualisierung wird verwendet, um Ressourcenkonflikte zu vermeiden. In einer virtuellen Maschine ist jeder Ressourcenbezeichner eindeutig. Derselbe Ressourcenbezeichner kann jedoch über *mehrere virtuelle Maschinen* hinweg, die auf *einer physikalischen*

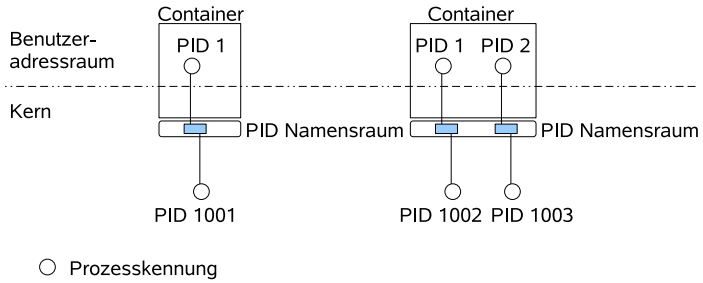

Abbildung 4.8: PID-Virtualisierung und -Isolation mit Namensraum

Maschine ausgeführt werden, mehrfach vergeben werden. Werden Prozess A und Prozess B aus obigem Beispiel jeweils in einer virtuellen Maschine ausgeführt, entsteht kein Ressourcenkonflikt.

Die unter Kapitel 1.3.2.1 aufgelisteten Techniken zählen zur sogenannten schwergewichtigen Virtualisierung, weil der Aufwand, ein vollständiges Betriebssystem in Form einer virtuellen Maschine, parallel zum Host-Betriebssystem und weiteren virtuellen Maschinen, auszuführen, groß ist, in Bezug auf Speicherverbrauch, Speicherverwaltung und Prozessornutzung.

4.3.2 Leichtgewichtige Virtualisierung mit Containern

Ein Container, auch bekannt als Virtual Private Server (VPS), beziehungsweise Virtual Environment, ist eine generische Technologie, um Ressourcen, die von mehreren Prozessen verwendet werden, voneinander zu isolieren und zu virtualisieren. Im Gegensatz zu Systemvirtualisierung anhand eines Virtual Machine Monitors (VMM) oder Hypervisors wird hierbei kein vollständiges Betriebssystem pro virtueller Maschine ausgeführt. Zudem ist keine Emulation auf Befehlssatzebene, noch Laufzeitkompilierung notwendig. Container können Maschineninstruktionen nativ, ohne spezielle Interpretationsmechanismen, auf dem Prozessor ausführen [72].

Container basieren auf leichtgewichtiger Virtualisierung. Während in früheren Linux-Versionen alle Kernressourcen pro Ressourcentyp in *einer globalen Tabelle* verwaltet wurden, existieren in neueren Versionen *Prozess-gebundene Namensräume* pro Ressourcentyp, [21]. Das heißt, Bezeichner von Prozessen, SYSV IPC Semaphoren, Segmenten, Nachrichtenwarteschlangen, Netzwerkadressen, sowie Rechnern und Benutzern werden mithilfe von Namensräumen vom Kern in den Benutzeradressraum abgebildet. Namensraumübergreifend können daher Ressourcen gleicher Kennung existieren, hierdurch wird *Ressourcenvirtualisierung* erzielt. Innerhalb eines Namensraumes besitzt jede Ressource einen eigenen Bezeichner, hierbei wird *Ressourcenisolation* realisiert, siehe Abbildung 4.8.

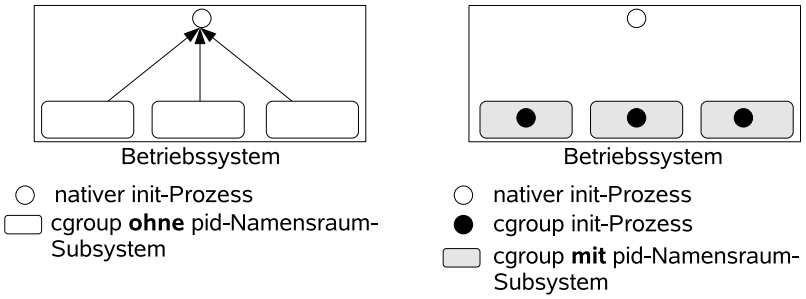

Abbildung 4.9: init-Prozess bei cgroups mit und ohne PID-Namensraum

4.3.2.1 Cgroup Subsysteme

Wie in Kapitel 4.2.4.2 bereits erwähnt, können *cgroups* verwendet werden, um beispielsweise Prozesse zu gruppieren. Die initiale *cgroup* enthält zunächst alle Prozesse eines Systems. Im Anschluss erzeugte *cgroups* sind hierarchisch angeordnet und enthalten jeweils Untermengen der Gesamtprozessmenge.

Die erweiterte Bedeutung von *cgroups* entsteht im Zusammenhang mit sogenannten Subsystemen [102]. Ein Subsystem ist ein Modul, welches einer *cgroup* hinzugeladen wird, um *cgroup*-spezifisches Verhalten zu realisieren. Pro *cgroup* können ein oder mehrere Subsysteme eingebunden werden. Ein Subsystem ist typischerweise ein Ressourcen-Controller, der bestimmte Ressourceneinschränkungen für eine Prozessgruppe definiert, wie beispielsweise Speicherverbrauch oder Prozessorzeit.

Ressourcenkonflikte werden vermieden, indem mit einer *cgroup* diverse Namensraum-Subsysteme assoziiert werden. In diesem Zusammenhang werden derzeit Ressourcen wie utsname-, Prozess-, IPC- und Benutzerkennungen sowie Netzwerkadressen, */proc*-Verzeichnisse et cetera unterstützt, siehe http://lxc.sourceforge.net/.

4.3.2.2 Zuverlässigkeit von cgroups

Jede *cgroup* hat eine Referenz auf einen *init*-Prozess. Bei der initialen *cgroup* und nachfolgend erzeugten *cgroups ohne* PID-Namensraum-Subsystem (PNS) entspricht er dem nativen *init*-Prozess des Betriebssystems, siehe linke Seite in Abbildung 4.9. Ein Prozess, dessen Vater terminiert, wird zum Prozesswaisen, der native *init*-Prozess wird zum neuen Vater des Waisen. Der *init*-Prozess existiert bereits vor dem Anführer einer UNIX-Sessiongruppe und damit auch UNIX-Prozessgruppe. Terminieren Anführer von UNIX-Session- und -Prozessgruppe, terminieren die verbleibenden Prozesse der Gruppe nicht. Hingegen entspricht der *init*-Prozess dem *ersten Prozess der cgroup*, wenn *cgroups* mit PNS konfiguriert werden, siehe rechte Seite in der Abbildung 4.9. Terminiert dieser *init* Prozess, terminieren alle Kinder automatisch. Das bedeutet insbesondere, dass insofern der *cgroup-init*-Prozess der Anführer einer UNIX-Prozessgruppe oder UNIX-Sessiongruppe ist

und terminiert, terminieren die verbleibenden *cgroup*-Prozesse ebenfalls. Dies stellt eine Verletzung der UNIX-Prozessgruppen- und -Sessiongruppen-Semantik dar.

Der Unterschied zwischen Linux mit PNS-aktivierten *cgroups* und Linux ohne *cgroups* muss bei der Prozessgruppenabbildung berücksichtigt werden. Prinzipiell erhöht sich daduch bei *cgroups* mit PNS die Wahrscheinlichkeit, dass eine verbleibende Anwendung ausfällt, nachdem der UNIX-Prozessgruppenanführer oder Sessionanführer terminiert hat.

Um diese Einschränkung zu beheben, muss der Linux Kernquelltext entsprechend angepasst werden.

4.4 Verwandte Arbeiten

In [101] wird beschrieben, wie auf anwendungstransparente Art und Weise Prozessereignisse in einem SSI-Betriebssystem erkannt und an ein darüberliegendes Grid-Computing-System weitergeleitet werden.

In [81] werden Jobs auf elementare Prozesse mithilfe sogenannter *execute-job-wrapper* und *recovery-job-wrapper* abgebildet. Diese Wrapper erfüllen im Wesentlichen die, in diesem Kapitel erwähnten, Aspekte der Job-Prozess-Assoziation. Ein Checkpointer-spezifischer *execute-job-wrapper* wird anstelle eines Jobs eingereicht. Er bildet die JobID auf jene Kennung ab, die von einem zugrundeliegenden Checkpointer verwendet wird. Der *recovery-job-wrapper* wird anstelle des Jobs beim Restart eingereicht. Analog zum *execute-job-wrapper* übergibt er die vom Checkpointer benötigten Restartinformationen. Er sorgt des weiteren für die neue Abbildung zwischen einem wiederhergestellten Job und dessen Prozessen. Die Vielfalt existierender Prozessgruppen, deren Eigenschaften und Checkpointerbezug werden nicht berücksichtigt. Desweiteren werden keine Informationen genannt, wie leichtgewichtige Virtualisierungstechnologien in eine Checkpointing-Architektur integriert werden können und welche Vorbereitungsmaßnahmen hierfür notwendig sind.

Zap verwendet sogenannte *pods*, Abstraktionen virtueller Maschinen, um pro Knoten Prozesse einer verteilten Anwendung zu gruppieren und sie von jenen des zugrundeliegenden Systems zu entkoppeln [85]. Pods verwenden *pod namespaces*, um Ressourcenvirtualisierung zu realisieren. Pods basieren darauf, Systemaufrufe abzufangen. *Pods* sind zap-spezifisch und daher in keinem nativen Linuxsystem integiert. Ähnlich wie zap verbindet das proprietäre IBM Metacluster eine Containertechnologie mit Checkpointing. Weil der zap- und der Metacluster-Quelltext nicht verfügbar sind, können insbesondere keine Angaben zum Prozessgruppenverhalten beim Verlust des Elternprozesses gemacht werden. Beide Container existieren unabhängig von einem Grid-Computing System. Die Weiterreichung von Prozessereignissen aus diesen Containern an das Job-Management eines darüberliegenden Grids wurde bisher nicht dargelegt.

4.5 Zusammenfassung

Ein Job ist eine Abstraktion des Grids, ein Prozess eine des nativen Beriebssystems. Prozesserzeugungen und Prozessterminierungen müssen auf Gridebene erkannt und auf Jobs korrekt abgebildet werden, damit Checkpoint-Anweisungen von der Gridebene aus zu konsistenten Checkpoints führen. In der GCA werden erstmalig relevante Prozessereignisse auf *anwendungstransparente* Art und Weise auf Gridebene erkannt. Die GCA setzt hierfür den Linux Process Event Connector ein, welcher Prozessereignisse im Kern erkennt und den überwachenden Prozess im Benutzeradressraum mithilfe einer Netlink-Verbindung darüber informiert. Die GCA stellt einen separaten Prozessereignis-Monitor bereit, um Checkpointer, die keine Netlink-Verbindungen erlauben, zu unterstützen.

Die meisten Checkpointer sichern ausschließlich Prozesse, die in Gruppen angeordnet sind. Hierzu zählen beispielsweise UNIX Prozessgruppen, UNIX Sessions, einfache Prozessbäume oder LinuxSSI-Anwendungsprozessgruppen. Ein bedeutender Aspekt ist hierbei, dass nicht jede Prozessgruppe von jedem Checkpointer unterstützt wird. Damit alle Prozesse einer Job-Einheit von einem Checkpointer korrekt gesichert und wiederhergestellt werden können, muss die vom Checkpointer unterstützte Prozessgruppe, als vorbereitende Massnahme für eine zukünftige Checkpointing-Aktion, bei Job-Einreichung erzeugt werden. Zum Checkpointing-Zeitpunkt muss die mit der Prozessgruppe assoziierte Kennung dem Checkpointer übergeben werden. Der Zusammenhang von UNIX-spezifischem Prozess-Management und Checkpointing wurde im Grid-Computing-Kontext bisher nicht, beziehungsweise bei CoreGRID ohne Bezug auf Prozess-reparenting, betrachtet.

Ressourcenkonflikte treten auf, wenn ein Ressource beim Wiederaufbau benötigt wird, eine gleichnamige, beispielsweise eine PID, jedoch bereits verwendet wird. Ressourcenkonflikte werden vermieden, wenn Checkpointer eingesetzt werden, die Prozesse in Containern gruppieren und Ressourcenvirtualisierung ermöglichen. Letzteres basiert bei den meisten Containertechnologien auf leichtgewichtiger Virtualisierung. Die GCA unterstützt Containerbasierte Checkpointer.

Für einen erfolgreichen Restart werden wichtige Metadaten, wie Prozessgruppenbezeichner und -typ sowie Containertyp und Containerkonfigurationen von der Übersetzungsbibliothek ermittelt und vom JEC abgespeichert.

5 Callbacks

Callbacks werden im Fehlertoleranzkontext eingesetzt, um Checkpoints und Restarts gezielt zu optimieren. In diesem Kapitel werden Grid-typische Anwendungsfälle von Callbacks, relevante Callback-Konzepte und deren Integration in die GCA dargestellt.

5.1 Anwendungsfälle

Callback-Funktionen können aus unterschiedlichen Motivationen heraus unmittelbar *vor und nach einem Checkpoint*, sowie *nach einem Restart* ausgeführt werden. Diese Funktionen werden im Folgenden als Pre-Checkpoint-, Post-Checkpoint- und Restart-Callbacks bezeichnet.

5.1.1 Optimierungen

Ein Kernel-Checkpointer kennt alle, ein Bibliothekscheckpointer kennt eine Untermenge der Strukturen, aus denen eine Anwendung besteht. Jedoch besitzt ein Checkpointer keine Kenntnis über die Semantik einer Anwendung. Das heißt, Checkpointern ist die Bedeutung von Dateien, Segmenten, et cetera einer Anwendung unbekannt.

Falls eine Anwendung in der Lage ist, jene Ressourcen zu kennzeichnen, die, aus ihrer Sicht, *nicht gesichert werden müssen*, kann Speicherplatz und Rechenzeit gespart werden. Aus Sicht einer Anwendung muss die dazugehörige Tera-Byte große Datenbank nicht bei jedem Checkpoint mitgesichert werden. Offensichtlich wird dadurch der Gesamtsicherungsaufwand überproportional erhöht.

Mithilfe eines Pre-Checkpoint-Callbacks können vor einer Sicherung entsprechende Ressourcen aus dem Anwendungskontext entfernt werden, beispielsweise indem der Dateideskriptor einer Datenbankdatei geschlossen wird. Nach einem Checkpoint kann diese Ressource im Post-Checkpoint-, beziehungsweise im Restart-Callback aufgebaut und anschließend erneut genutzt werden.

5.1.2 Komplementierung

Nicht jeder Checkpointer ist aus funktionaler Sicht in der Lage, alle möglichen Anwendungsressourcen zu sichern und wiederherzustellen, siehe Kapitel 1.3.1.6. In diesem Zusammenhang können Checkpointer oder das System verklemmen oder sogar abstürzen, was vermieden werden muss. Auch hier können Callbacks verwendet werden, um Checkpointer-Funktionalitäten zu komplementieren, indem die Sicherung und Rekonstruktion bestimmter Ressourcen in Callbacks verlagert wird. Der BLCR und LinuxSSI Checkpointer kann beispielsweise keine TCP Sockets sichern und wiederherstellen, siehe Kapitel 1.3.1.6. Werden bestimmte Ressourcen mithilfe von Callbacks vor dem Checkpointing abgebaut und nach dem Restart im Callback wiederaufgebaut, erhöht sich der Einsatzbereich eines Checkpointers.

5.1.3 Heterogenität

Bei Grid-Checkpointing müssen heterogene Checkpointer miteinander kooperieren. Hierbei stellen Callbacks ein allgemein nützliches Konzept dar. Beispielsweise kann hiermit vor und nach der verteilten Sicherung durch mehrere Checkpointer ein gemeinsamer Mechanismus, beziehungsweise ein gemeinsames Protokoll ausgeführt werden. Dadurch können Gridknoten-übergreifende Ressourcen, wie Nachrichtenkanäle, siehe Kapitel 6, Griddateisystem-Dateien, gemeinsam genutzte Segmente, Pipes konsistent sichern und wiederherstellen.

5.2 Callback-Architekturaspekte

5.2.1 Kritische Abschnitte

Checkpointaufforderungen können zu ungünstigen Zeitpunkten auftreten. Ist ein Checkpointer beispielsweise nicht in der Lage, Sockets zu sichern und kommt der Checkpointaufruf vor der Registrierung jenes Callbacks, welcher Sockets sichert und abbaut, schlägt die Checkpointoperation fehl. In [43] werden Quelltext-Fragmente einer Anwendung als kritische Abschnitte deklariert, um sie atomar auszuführen, sodass keine Unterbrechung aufgrund einer Checkpointing-Operation auftreten kann.

5.2.2 Signal- versus Thread-Kontext

Callbacks können im Signal-Handler-Kontext ausgeführt werden. Hierbei können jedoch Verklemmungen, infolge fehlender Synchronisierung, auftreten. Wird beispielsweise eine

Sperre von einem Anwendungsthread aquiriert und anschließend der Signal-Handler ausgeführt, erhält letzterer die höchste Priorität. Versucht der Signal-Handler selbst die Sperre anzufordern, wird er blockiert. Weil der Signal-Handler erst nach Behandlung des Signals zurückkehren kann, entsteht eine Verklemmung.

Dieses Problem wird vermieden, wenn Signal-sichere Funktionen[1] verwendet werden. Ein Programmierer maskiert alle Signale für den Zeitraum der Funktionsausführung, sodass Signal-Unterbrechungen und deren genannte Auswirkungen verhindert werden [78]. Hierdurch entsteht jedoch eine zusätzliche Belastung des Programmierers, wenn er Signale explizit maskieren muss. Zudem kommt es hierbei zu Performanzeinbußen.

Werden Callbacks im Thread-Kontext ausgeführt, entstehen keine Verklemmungen, weil kein gegenseitiges Warten auftritt.

Damit Datenkonsistenz, bei gleichzeitiger Ausführung von Anwendungs- und Callback-Threads, gewährleistet wird, muss der Zugriff auf gemeinsam genutzte Daten explizit synchronisiert werden[2]. BLCR empfiehlt beispielsweise Bibliotheksautoren bibliotheksweite Sperren zu integrieren, die vom einzigen BLCR Callback-Thread aquiriert werden können und Anwendungsthreads bei erneutem Aufruf einer Bibliotheksfunktion blockieren. Diese Vorgehensweise ist sicher, wenn keine C-Bibliotheksinternen Sperren bereits vor dem Eintritt in den Bibliotheksaufruf angefordert wurden.

5.3 Callback-Integration

5.3.1 Anwendungsmodifizierung

Falls ein Checkpointer keine eigene Callback-Infrastruktur besitzt, werden Callbacks mithilfe der GCA registriert und von ihr ausgelöst. In diesem Fall wird einer Job-Einheit die *gca.a*-Bibliothek, siehe Kapitel 3.5.1, statisch hinzugelinkt. Ein Callback wird ausgelöst, indem die Übersetzungs- und *gca.a*-Bibliothek miteinander interagieren. Da beides Elemente der GCA sind, müssen sich beide nicht zusätzlich auf einen gemeinsamen Kommunikationsmechanismus einigen, damit ein Callback von der Übersetzungsbibliothek aus bei der *gca.a*-Bibliothek ausgelöst werden kann.

Zusätzlich kann die *gca.a*-Bibliothek angewiesen werden, Callback-Funktionen einerseits *einheitlich* und andererseits bei der *Callback-Infrastruktur eines zugrundeliegenden Checkpointers* zu registrieren, um von deren Besonderheiten zu profitieren. Hierzu zählt, dass kritische Abschnitte berücksichtigt und Callbacks nebenläufig ausgeführt werden können.

[1] POSIX weist asynchrone signalsichere Funktionen aus, jedoch gehören Heap-Funktionen, wie *malloc* und *free*, nicht dazu.

[2] Die Stilllegung aller Anwendungthreads während der Callback-Ausführung stellt keine Option dar, da Deadlocks entstehen können, wenn ein Callback-Thread Zugriff auf eine Ressource erwünscht, welche von einem schlafen gelegten Anwendungsthread zuvor gesperrt wurde.

Callbacks werden beispielsweise bei BLCR und LinuxSSI mithilfe einer Checkpointerbibliothek in die Anwendung eingebunden. Weil sich bei Checkpointern wie BLCR und LinuxSSI nach der Callbackausführung die Prozesssynchronisierung und -sicherung unmittelbar anschließt, jedoch für verteiltes, koordiniertes Checkpointing anderem die Teilsequenzen *stop*, *checkpoint*, *resume_cp* notwendig sind, müssen explizit Barrieren eingebaut werden. Hierdurch werden Callbacks bei allen Checkpointer ausgeführt, bevor die Prozesssynchronisierung startet.

Um die Callback-Ausführung gezielt zu adressieren, muss die Übersetzungsbibliothek mit der Checkpointerbibliothek kommunizieren. Beide Bibliotheken können mithilfe von Nachrichtenwarteschlangen, gemeinsam genutzten Speichersegmenten, UNIX Sockets oder Signalen miteinander kommunizieren, deren Identitäten auf beiden Seiten bei der Callback-Registrierung ausgehandelt werden müssen. Der Anwendungsprogrammierer bestimmt bei der Callback-Registrierung die optimale Kommunikationsart, sodass beispielsweise keine Signal verwendet wird, welches die Anwendung bereits einsetzt.

5.3.2 Transparente Callback-Integration und Library Interposition

Neben der expliziten Callback-Registrierung, bei der die Anwendung modifiziert wird, können Callbacks auch anwendungstransparent eingebunden werden. Hierzu wird der

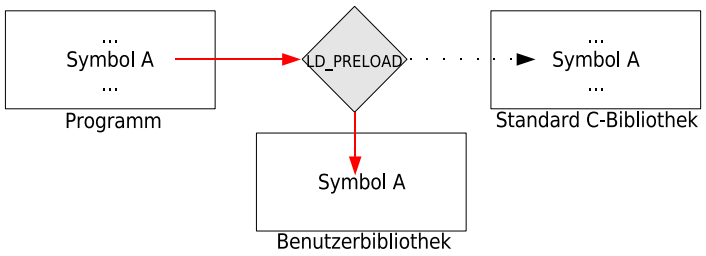

Abbildung 5.1: LD_PRELOAD Mechanismus

Linux-spezifische LD_PRELOAD Mechanismus eingesetzt, um Library-Interposition, beziehungsweise Library-Interception, Bibliotheksaufrufe werden abgefangen, zu erzielen. In Abbildung 5.1 wird dessen Funktionsweise dargestellt. Aus Sicht des Programms implementieren die Benutzerbibliothek und die Standard C-Bibliothek ein Symbol mit identischer Signatur. Wurde die Umgebungsvariable LD_PRELOAD vor dem Programmstart mit dem Pfad zur Benutzerbibliothek initialisiert, führt der Symbol A-Aufruf des Programmes dazu, dass die Symbol A-Implementierung der Benutzerbibliothek, anstelle der Standard C-Bibliothek, adressiert wird. Die Benutzerbibliothek kann anschließend, optional, Symbole anderer Bibliotheken adressieren.

Ein Callback kann *anwendungstransparent* bei Kindprozessen registriert werden, wenn ein sogenannter *fork*-Wrapper eingesetzt wird. Dateiauszug 5.1 stellt den Ausschnitt einer Benutzerbibliothek dar, die mit dem LD_PRELOAD Mechanismus der Standard C-

Bibliothek vorgeladen wird. Während die Anwendung in gewohnter Art und Weise *fork* aufruft, wird *_real_fork* in der Benutzerbibliothek adressiert. Darin wird zunächst die *fork* Funktion der Standard C-Bibliothek gerufen und danach eine Callback-Funktion beim BLCR-Checkpointer registriert[3], ohne dass die Anwendung darüber informiert wird.

Um abzusichern, dass jeder neue Prozess Callbackfunktionen in anwendungstransparenter Art und Weise registriert, kann neben dem *fork*-Wrapper auch ein *clone*-Wrapper verwendet werden, wenn dabei zwischen Thread- und Prozesserzeugung differenziert wird.

Listing 5.1: *fork*-Wrapper

```
#define CALL_REAL_SYS(name, ret, sig) \
        static ret (*func) sig = NULL; \
        if (!func) { \
                func = (ret (*) sig) dlsym(RTLD_NEXT, #name);\
                if (func == NULL) { \
                        //dlerror\
                } \
        } \
        return func

pid_t _real_fork(){
        CALL_REAL_SYS(fork, pid_t, ())();
        //Checkpointer-spezifische Callback-Registrierung
        cr_init();
        cr_register_callback(   &cb_rebuild_sockets,
                                NULL,
                                CR_THREAD_CONTEXT);
}
```

Die unter Kapitel 3.5.1 beschriebene *shared-resources.so* Bibliothek wird auf diese Art und Weise eingebunden, um Callbacks transparent einzubinden, um gemeinsam genutzte Ressourcen, oberhalb heterogener Checkpointer, behandeln zu können.

[3]Damit die BLCR-spezifischen Callback-Routinen aufgelöst werden können, muss die Benutzerbibliothek gegen die BLCR-Bibliothek gelinkt sein.

5.4 Verwandte Arbeiten

Bislang existieren nur sehr wenige Checkpointer-Pakete, die Callbacks unterstützen. In BLCR können Callbacks im Signalhandler- als auch im Thread-Kontext ausgeführt werden. Zudem können kritische Abschnitte markiert werden [43]. Bei der Kombination von LAM MPI mit BLCR werden Callbacks eingesetzt, um MPI-Kommunikationskanäle vor einem Checkpoint zu leeren [125].

Beim LinuxSSI Kernel Checkpointer können Callbacks im Thread-Kontext ausgeführt werden, jedoch fehlen Konstrukte, um Verklemmungen auszuschließen [49]. In [28] wird erwähnt, dass der ckpt Checkpointer Callbacks unterstützt.

Zudem exisitert bisher kein Grid-Checkpointer, mit dessen Hilfe generische und benutzerdefinierte Callbacks bei der Anwendung registriert werden können.

5.5 Zusammenfassung

Callbacks sind wichtig im Checkpointing-Kontext, weil sie anwendungsgesteuerte Optimierungen beim Checkpointing ermöglichen und funktionelle Beschränkungen eines Checkpointers transparent komplementieren.

In dieser Arbeit wurden Einsatzmöglichkeiten von Callbacks im Checkpointing-Kontext vorgestellt. Hierbei werden Callbacks verwendet, um einen gridknotenübergreifenden Mechanismus bereitzustellen, mit dem gemeinsam genutzte Ressourcen, wie beispielsweise Dateien und Kommunikationskanäle gesichert und wiederhergestellt werden können. In der GCA werden erstmalig Callbacks verwendet, um *heterogene Checkpointer*, in Bezug auf die erwähnten Ressourcen, *transparent miteinander kooperieren zu lassen*.

In diesem Kapitel wurde dargelegt, dass die GCA eine eigene Callback-Infrastruktur bereitstellt, anhand derer Callbacks in *uniformer* Art und Weise, unabhängig von einem zugrundeliegenden Checkpointer, registriert werden können. Callbacks werden dann entweder im Kontext existierender Checkpointer-Pakets-Bibliotheken oder der GCA-Bibliothek ausgeführt.

Des Weiteren wurde der LD_PRELOAD-Mechanismus und der sogenannte *fork*-Wrapper vorgestellt, anhand derer Callbacks *anwendungstransparent* registriert werden können. Hierdurch lässt sich eine *generische Kanalsicherung* für nachrichtenbasierte Anwendungen elegant realisieren, siehe Kapitel 6.

6 Gridkanalsicherung

Über Nachrichten kommunizierende Anwendungen im Grid können nur dann konsistent gesichert und wiederhergestellt werden, wenn die Gridknoten-übergreifenden Kommunikationskänale mit einbezogen werden. Die Behandlung von in-Transit Nachrichten ist dabei von zentraler Bedeutung. Eine besondere Herausforderung besteht darin, Kanäle auf Basis heterogener Checkpointer entsprechend zu sichern und zu rekonstruieren.

Zunächst wird Gridkanal-Management hinsichtlich existierender Checkpoint- und Kommunikations-Protokolle, sowie der GCA-Kriterien hin untersucht. Anschließend wird die GCA-Gridkanalsicherung ausführlich erläutert.

6.1 Problemeingrenzung

6.1.1 Behandlung von in-Transit Nachrichten

Während der fehlerfreien Ausführung auftretende in-Transit-Nachrichten können, ohne entsprechende Operationen zum Checkpoint-Zeitpunkt, beim Restart-Zeitpunkt dazu führen, dass verwaiste oder verlorene Nachrichten auftreten, siehe Kapitel 1.2.1.

Eine Nachricht gilt als verloren, wenn das Sendeereignis Teil des Senderprozess-Checkpoints, das Empfangsereignis jedoch nicht Teil des Empfängerprozess-Checkpoints ist. Wird mithilfe dieser beiden Checkpoints ein Neustart ausgeführt, wird die Nachricht *nicht mehr an den Empängerprozess ausgeliefert*, was inkonsistent ist.

Eine Nachricht gilt als verwaist, wenn das Sendeereignis nicht Teil des Checkpoints, des, in Zukunft sendenden, Prozesses ist, jedoch das zugehörige Empfangsereignis Teil des Empfängerprozess-Checkpoints ist. Bei einem Neustart, basierend auf diesen beiden Checkpoints, wird die Nachricht *erneut an den Empfänger ausgeliefert*, was inkonsistent ist.

Im Folgenden wird betrachtet, wie in-Transit Nachrichten bei verschiedenen Checkpointing- und Kommunikations-Protokollen verwaltet werden, um die Problemstellung im Grid-Checkpointing-Kontext genauer zu identifizieren.

6.1.1.1 Checkpointing-Protokolle

Weil bei *unabhängigem Checkpointing* die beteiligten Anwendungsprozesse nicht miteinander synchronisiert werden, kann das Sende- und Empfangsereignis einer Nachricht in zwei lokalen Checkpoints vorhanden sein, die nicht zu einem globalen Checkpoint gehören. Da erst zum Zeitpunkt des Restarts mithilfe zuvor aufgezeichneter Abhängigkeitsinformationen ein konsistentes Abbild ermittelt wird, müssen keine Kanal-Inhalte zum Checkpoint-Zeitpunkt gesichert werden.

Beim Checkpointingprotokoll nach [29] werden nebenläufig Sicherungen durchgeführt. Verlorene Nachrichten werden vermieden, indem eine spezielle Marker-Nachricht in-Transit Nachrichten aus FIFO Kanälen herausschiebt und in einem lokalen Checkpoint abspeichert. Verwaiste Nachrichten werden bei versetzt auftretenden Sicherungen verschiedener Prozesse vermieden, indem ein Marker, auf dem gleichen Kanal eintreffende, Anwendungsnachrichten einem lokalen Checkpoint zuordnet, der zu einem global konsistenten Checkpoint gehört. Inwiefern dieses Protokoll im Grid-Checkpointing-Kontext angewendet werden kann, wird unter 6.7 diskutiert.

Bei nicht-nebenläufigem koordinierten Checkpointing können keine verwaisten Nachrichten auftreten. Zunächst werden alle verteilten Anwendungsprozesse synchronisiert beziehungsweise stillgelegt. Nach dem logischen Zeitpunkt T_x wird ein globaler Checkpoint erzeugt. Zu einem späteren logischen Zeitpunkt T_{x+1} fahren alle Prozesse mit ihrer Ausführung fort. Diese sequentielle, für alle Prozesse gültige, Vorgehensweise bewirkt, dass *nach* T_{x+1} ausgesandte Nachrichten von keinem Prozess *vor* dem Checkpoint empfangen werden können. Hingegen können verlorene Nachrichten auftreten, wenn im Kanal befindliche Nachrichten nicht als Empfangsereignisse im Empfänger-Checkpoint integriert werden. *Zum Checkpoint-Zeitpunkt sind daher explizit entsprechende Maßnahmen notwendig.*

6.1.1.2 Kommunikations-Protokolle

Um konsistente Zustände zu sichern, können die hierzu abzusichernden Fälle weiter reduziert werden, wenn existierende Kommunikationsprotokolle einbezogen werden. Auf Transportschicht-Ebene des Netzwerkprotokollstacks werden zwei Protokollklassen der Nachrichtenauslieferung unterschieden: zuverlässige, beispielsweise TCP, und unzuverlässige, beispielsweise UDP, Protokolle. Protokolle der unzuverlässigen Nachrichtenauslieferung garantieren inhärent keine erfolgreiche Nachrichten-Zustellung. Somit sind verlorene oder duplizierte Nachrichtenpakete vergleichbar mit Kommunkationsfehlern, welche von dieser Protokollklasse ohnehin nicht behandelt werden. Deshalb müssen unzuverlässige Protokolle beim Checkpointing nicht berücksichtigt werden.

Zuverlässige Kommunikationsprotokolle hingegen *sichern ab, dass Nachrichten während der fehlerfreien Ausführung ausgeliefert werden.* Letzteres gilt nicht im Kontext von Kno-

ten-, Netzwerk-, beziehungsweise Prozessausfällen.[1]

Letzteres erfordert, dass *Rollback-Recovery oberhalb der Transportschicht, ausschließlich für zuverlässige Nachrichtenprotokolle*, umgesetzt werden muss.

6.1.2 Kooperation heterogener Checkpointer

Weil sich die Sockets eines Kommunikationskanals auf zwei verschiedenen Gridknoten befinden können, können bei koordiniertem Checkpointing in-Transit Nachrichten nur dann entsprechend behandelt werden, wenn heterogene Checkpointer miteinander kooperieren. Kooperieren Checkpointer C_A auf Gridknoten A und Checkpointer C_B auf Gridknoten B nicht miteinander, ist unklar:

- woher C_A weiß, zu welchem Knoten ein wiederherzustellender Client-Socket beim Restart verbunden werden soll. Der Server-Socket kann zwischenzeitlich migriert worden sein und
- woher C_A die Semantik von Protokollnachrichten kennt, die C_B zur Kanalzustandsicherung sendet. Sie können Anweisungen für das Verhalten C_A's enthalten.

Da ein Checkpointer andere Checkpointer und deren Funktionalitäten und Semantiken nicht kennt, muss die GCA Kooperations-fördernde Maßnahmen und Komponenten bereitstellen.

6.2 GCA-Kriterien und Gridkanal-Management

TCP wird im Kern durch umfangreiche Datenstrukturen repräsentiert, wie beispielsweise

- Endpunkt-Identitäten (IP-Adresse und Port),
- Fenster für die Flusskontrolle (snd_wnd, rcv_wnd),
- Sequenznummern (rcv_nxt, snd_nxt),
- Zeitstempel (tcp_time_stamp),
- Puffer (bereit für die Auslieferung, out-of-order, zu sendende, erneut zu sendende Daten),
- Verbindungszustand,
- selektive Bestätigungen, etc.

[1] TCP gewährleistet Verbindungs-Funktionalität, wenn kurze Unterbrechungen im Sekundenbereich auftreten. Dies stellt jedoch keine generische Lösung, für Ausfälle von mehreren Stunden oder Tagen, dar.

Für die in Kapitel 6.2.3 dargestellten Kriterien einer Gridkanalsicherung ist erheblich, in welchem Umfang welche TCP-Daten aus dem Kern extrahiert werden müssen und inwiefern Änderungen am Betriebssystem notwendig sind.

Nachrichtenkanalzustände werden im Kontext von Lastverteilung und Fehlertoleranz auf unterschiedliche Art und Weise behandelt. Sie gleichen sich jedoch in dem Aspekt, dass mindestens ein Kanalende wiederhergestellt werden muss. Bei Lastverteilung wird dabei ein Kanalende von einem fehleranfälligen auf einen stabileren Knoten übertragen, ohne dass die Verbindung unterbrochen wird. Bei Fehlertoleranz muss ein Kanal wiederhergestellt werden, insbesondere dann, wenn er beispielsweise infolge von Rechnerausfällen zuvor zerstört wurde.

Beide Anätze und deren Implikationen werden nachfolgend kurz skizziert, um die für Grid-Checkpointing relevante Lösung zu ermitteln.

6.2.1 Ansatz 1 - einseitige Kanalendpunktsicherung

Beim sogenannten Live-Migrationsansatz wird ein Kanalende migriert, wobei die Verbindung temporär stillgelegt wird, ohne sie zu schliessen. Während der Sicherungsdauer beider Peer-Prozesse und der Wiederherstellung des Sockets auf dem Zielrechner werden keine Nachrichten ausgesendet, beziehungsweise bereits empfangene Nachrichten nicht bestätigt. Damit keine Nachrichten verloren gehen, wird der TCP-interne Fehlertoleranzmechanismus der Neuübertragung bisher nicht bestätigter Pakete (engl. TCP retransmission) eingesetzt.

TCP Verbindungen besitzen einen sogenannten *keep alive Zeitgeber*. Demnach wartet ein Sender ein Zeitinterval Z darauf, dass der Empfänger ein eingetroffenes Paket beim Sender bestätigt. Die Verbindung wird geschlossen, wenn Z überschritten wurde.

Da Prozesssicherungen jedoch länger dauern können als Z muss der TCP *keep alive Zeitgeber* bei stillgelegten Verbindungen manipuliert werden (Timer verhandeln oder deaktivieren), damit die Verbindung auch über die Spanne der Prozesssicherung hinaus aktiv bleibt. Letzteres wird als *TCP session preservation* bezeichnet. Hierzu muss die Transportschicht des Kerns abgeändert werden.

Eine weitere Live-Migration-spezifische Herausforderung besteht darin, Pakete vom alten zum neuen Empfänger-Socket zu leiten. Hierfür existieren unterschiedliche Verfahren, beispielsweise können Tunnel angelegt oder Network Address Translation (NAT) eingesetzt werden. In [89] wird eine Cluster-IP-Adresse auf eine IP-Addresse einer Multicastgruppe abgebildet. Anschliessend muss jeder Knoten der Multicastgruppe autark entscheiden, ob er eine neue Verbindung annimmt. Bei diesem Verfahren muss das Address Resolution Protocol (ARP) abgeändert werden.

Des weiteren ist eine Anpassung des tcp_time_stamp Zählers im Kern notwendig, welcher lokale Zeitstempel erzeugt. Werden die Zähler eines Quellrechners A und eines Zielrechners B nicht synchronisiert, wird die lokale round-trip-time (RTT) Berechnung gestört, da der

Zeitstempel einer von A ausgesendeten Nachricht mit deren Bestätigung zum zwischenzeitlich zu B migrierten Sender zurückgesandt wird. Ein falscher RTT-Wert kann Ursache für einen verminderten Verbindungsdurchsatz sein.

Neben den erwähnten Kernänderungen muss eine Schnittstelle im Kern implementiert werden, um Datenstrukturen wie diverse Puffer, Zeitstempel und Sequenznummern eines TCP-Endpunktes auf Kern-Ebene herauszuschreiben und wiederherzustellen.

6.2.2 Ansatz 2 - beidseitige Kanalendpunktsicherung

Eine Verbindung wird unterbrochen, wenn beispielsweise die Knoten beider Kanalenden zusammen ausfallen. Im Gegensatz zum Live-Migrationsansatz kann in diesem Szenario ein Kanal nur dann konsistent wiederhergestellt werden, wenn vor dem Ausfall relevante TCP-Daten, wie in-Transit-Nachrichten, extrahiert und *persistent gesichert* worden sind. Wenn beide Kanalenden ausfallen, kann keine TCP Session Preservation eingesetzt werden, deren Voraussetzung ein funktionierendes Kanalende ist.

Hierdurch entfallen die unter 6.2.1 erwähnten Kernänderungen[2]. Anstelle dessen können die vorhandenen Netzwerkschnittstellen zum Kern verwendet werden, um den Kanal wiederherzustellen.

6.2.3 Synthese

Eine Gridkanalsicherungslösung muss folgenden Aspekten genügen:

1. Anwendungstransparenz: in-Transit Nachrichten müssen ohne Benutzerinteraktion behandelt werden und ein Programmierer soll keine eigene Kanalsicherungs-Funktionalität entwickeln müssen.
2. Betriebssystemtransparenz: die Verbindungs-, Netzwerk- und Transport-Schichten der Netzwerkprotokollarchitektur sollen nicht abgeändert werden.
3. Checkpointertransparenz: Checkpointer-Kooperation darf nicht erzielt werden, indem letztere modifiziert werden.
4. Unterstützung heterogener Checkpointer: auch Checkpointer, die von sich aus keine Sockets sichern/wiederherstellen können, sollen einbezogen werden können.
5. Unterstützung von Anwendungsmigration: sich ändernde Adressdaten beteiligter Kanalenden dürfen die Kanalwiederherstellung nicht beeinträchtigen.

[2]Wenn das vor Checkpointing gültige TCP-Sendefenster (snd_nxt), das TCP-Empfangsfenster (rcv_nxt), der Fenstergrößenänderungsfaktor, etc. beim Restart nicht wiederhergestellt werden, kann der Verbindungsdurchsatz beeinträchtigt werden.

6. Effizienz der in-Transit Nachrichten-Behandlung. In-Transit Nachrichten müssen gesichert und wiedereingespielt werden, ohne die Anwendung umfassend zu beeinflußen.

Weil zwei Gridknoten, die über einen Kanal miteinander verbunden sind, gleichzeitig ausfallen können, müssen all jene Daten extrahiert und gesichert werden, anhand derer eine vormals unterbrochene Verbindung wiederhergestellt werden kann. Ansatz 1 ist ungeeignet, weil einerseits das Kriterium eines aktiven Kanals nicht immer im Grid eingehalten werden kann. Andererseits kollidieren die beschriebenen Kernänderungen, um Live-Migration zu realisieren, mit dem Kriterium der Betriebssystemtransparenz. Im Folgenden wird daher Ansatz 2 verfolgt, bei dem alle hier aufgelisteten Kriterien erfüllt werden.

6.3 Gridkanalsicherung im Überblick

Die Strategie der Gridkanalsicherung (GKS) besteht darin, in-Transit-Nachrichten aller TCP-Kanäle einer verteilten Anwendung vor dem Checkpointzeitpunkt herauszuschieben, sodass bei koordiniertem Checkpointing keine Nachrichten verloren gehen können.

Hierzu werden alle Anwendungsthreads während der Sicherungsoperation daran gehindert, Nachrichten zu senden beziehungsweise zu empfangen, zusätzlich dürfen keine weiteren Kanäle erzeugt werden. Letzeres wird durch separate Kontroll-Threads erzielt, welche Kanäle leeren und Nachrichten erneut einspielen. Nach der Installation eines Kontroll-Threads pro Kanal-Prozess, wird eine Marker-Nachricht vom Sender zum Empfänger gesendet, welche dem Empfänger signalisiert, dass der Kanal leer ist. Potentielle in-Transit Nachrichten werden auf Empfängerseite in einem Kanalpuffer zwischengespeichert.

GKS unterstützt Checkpointer, die offene Socketdeskriptoren nicht sichern und wiederherstellen können, indem diese Deskriptoren vor einem Checkpoint optional geschlossen und nach einem Checkpoint, beziehungsweise Restart rekonstruiert werden. Bevor die Anwendung weiterrechnen darf, erfolgt die Deblockierung kanalerzeugender, nachrichtenversendender und -empfangender Aufrufe. Im Kanalpuffer zwischengespeicherte Nachrichten müssen von den Anwendungsthreads verarbeitet werden, bevor neue Nachrichten empfangen werden.

Bei GKS werden zudem veränderte IP-Adressen von Serversockets unterstützt, um die dynamische Migration von Job-Einheiten zu ermöglichen.

GKS wird in Form der dynamischen Bibliothek *shared_resources.so*, als Teil der UCS-AS, implementiert, siehe Kapitel 3.5.1.

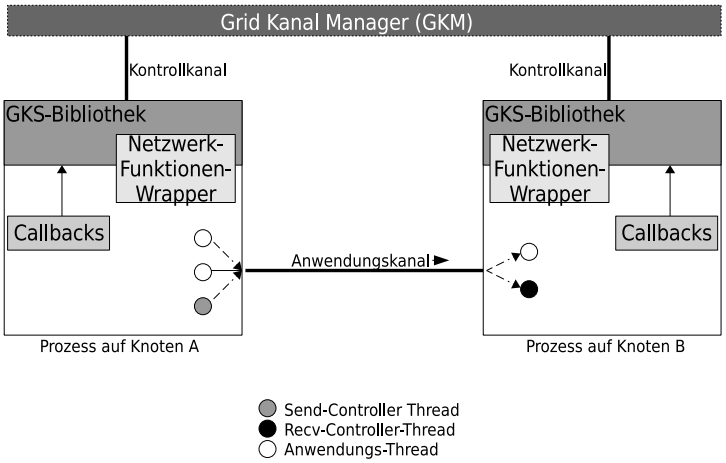

Abbildung 6.1: GKS-Komponenten

6.4 GKS-Architektur

Die GKS wird durch mehrere Komponenten und Konzepte realisiert, welche in Abbildung 6.1 dargestellt sind.

6.4.1 Überwachung von Netzwerk-Bibliotheksaufrufen

Für eine anwendungstransparente Kanalsicherung müssen Netzwerk-Bibliotheksaufrufe überwacht und gegebenenfalls GKS-spezifisch manipuliert werden. Anwendungsthreads dürfen weder Nachrichten senden (*send*), beziehungsweise empfangen (*recv*), noch neue Kanäle erzeugen (*socket*), während eine Sicherung abläuft. Im Gegensatz dazu müssen die Threads der GKS-Architektur GKS-Kontrollnachrichten durch Anwendungs- und Kontrollkanäle austauschen können.
Ferner muss die GKS bestehende, zuverlässige Anwendungskanäle kennen, um eine Kanalsicherung initiieren zu können. Die GKS markiert hierzu TCP Sockets, welche durch Aufrufe von *accept* und *connect* identifiziert werden. Darüber hinaus vermerkt die GKS, in welchem Modus ein Socket ist - er gibt Aufschluss über die Richtungen, in welche Marker gesendet werden müssen. Werden Daten empfangen, ist der Socket im *recv* Modus, werden Daten gesendet, ist er im *send* Modus. Ein Socket ist im Duplex Modus, wenn Datenverkehr in beide Richtungen getätigt wird, beispielsweise wenn ein Prozessthread sendet, ein anderer auf Daten am gleichen Socket wartet.

Anwendungstransparentes Abfangen von Netzwerk-Bibliotheksaufrufen kann erzielt werden, indem zwischen der Anwendung und der Netzwerkbibliothek (Standard C Bibliothek) die GKS-Bibliothek eingefügt wird, mithilfe des LD_PRELOAD Mechanismus, siehe Kapitel 5.3.2. Beispielsweise wird der *socket*-Aufruf zunächst an die GKS-Bibliothek ge-

Netzwerkfunktion	Bedeutung
socket	Erstelle Kommunikationsendpunkt.
bind	Assoziiere lokale Adresse mit Socket.
listen	Erwarte (begrenzte Zahl an) Socketverbindungen.
connect	Verbinde zu einem Socket.
accept	Akzeptiere neue Verbindung zu einem Socket.
send	Sende eine Nachricht.
recv	Empfange eine Nachricht.
setsockopt	Definiere Socketoptionen.

Tabelle 6.1: Abgefangene Netzwerkaufrufe

leitet, welche GKS-spezifische Daten ermittelt und anschließend die *socket*-Funktion der Netzwerkbibliothek aufruft. Tabelle 6.1 stellt alle Netzwerkaufrufe, die für GKS abgefangen und modifiziert werden müssen, dar.

6.4.2 Callbacks als Grundlage der GKS-Ausführung

Wird ein Server-Socket migriert, muss sich ein Client-Socket beim Restart zur neuen Adresse verbinden. Können offene Socket-Deskriptoren nicht gesichert werden, müssen sie vor dem Checkpoint anwendungstransparent geschlossen und später wiederhergestellt werden. Beide Aspekte können mithilfe von Callbacks realisiert werden, die vor und nach einem Checkpoint als auch nach dem Restart ausgeführt werden.

In der GCA können Callbacks, unabhängig vom zugrundeliegenden Checkpointer, registriert und ausgeführt werden, siehe Kapitel 6.1.2. Wird dieselbe Callback-Funktionalität auf mehreren Gridknoten ausgeführt, können mehrere lokale Checkpointing-Vorgänge miteinander kooperieren und somit GKS realisieren.

6.4.3 Kanalkontrollthreads und Kanal-Manager

Pro Prozess existieren zwei Protokollthreads, der send-Kontrollthread und der recv-Kontrollthread. Einer der beiden kann *Kanal-Manager (KM)* eines Kanals werden, abhängig vom Socket-Modus und der Socketnutzung, siehe weiter unten. Ein KM überwacht exklusiv das Leeren und Wiederherstellen eines Kanals, insbesondere wenn ein Kanal von mehreren Anwendungsthreads, eines oder mehrerer Prozesse, gemeinsam genutzt wird. Ist der Socket im Duplex Modus, wird ein send-Kontrollthread KM.

Der send-Kontrollthread verwaltet Sockets des send-Modus. Zum Checkpoint- und Restartzeitpunkt initiiert er *aktiv* die Kanal-Leerung und Wiederverbindung, in Kooperation mit dem entfernten recv-Kontrollthread.

Der recv-Kontrollthread verwaltet Sockets des recv-Modus'. In *passiver* Form reagiert er auf Anfragen eines entfernten send-Kontroll-Threads, einen Kanal zu leeren und erneut zu verbinden.

Die Aufteilung in send/recv-Kontrollthread ermöglicht eine nebenläufige GKS-Ausführung. Dies trifft für Situationen zu, in denen zwei Prozesse die Server- und Client-Rolle, für verschiedene Kanäle, gleichzeitig ausüben. Zwei, auf unterschiedlichen Gridknoten befindliche, Server-Sockets, die, jeweils von einem send-Kontroll-Thread, erzeugt wurden, blockieren einander nicht, wenn sie auf die Wiederverbindung des jeweils gegenüberliegenden Knotens warten. Anstelle dessen behandelt der jeweilige recv-Kontrollthread die Wiederverbindungsanfrage. Verklemmungen können in diesem Kontext verhindert werden.

6.4.4 Grid-Kanal-Manager

Wurde ein Server-Socket zwischenzeitlich migriert, scheitert ein Wiederverbindungsversuch eines Client-Sockets, wenn die alte IP-Adresse verwendet wird. Der Grid-Kanal-Manager (GKM) muss daher insbesondere Kanalmigrationen unterstützen.

Der GKM vergibt eine gridweit eindeutige Socket-ID, wenn ein Socket initial erzeugt wird. Die Socket-ID und der erste mit dem Socket assoziierte Socketdeskriptor werden immer im Checkpoint abgespeichert, auch wenn Sockets vor dem Checkpointing geschlossen werden, da sie für eine korrekte Socketwiederherstellung und erneute Registrierung beim GLM erforderlich sind. Wird beispielsweise ein Socket nach einem Restart rekonstruiert, wird die zuvor gültige Socket-ID eines Socketdeskriptors aus dem Checkpoint gelesen und dem GKM zur erneuten Registrierung übergeben. Der GKM verwaltet ein Socket, indem er jeweils die Socket-ID mit der temporären Socket-IP-Adresse abspeichert.

Darauf aufbauend registriert der GKM einen logischen Kanal, wenn zwei miteinander in Beziehung stehende Sockets erkannt wurden. Jeder Kanal wird durch einen eindeutigen, über seine Lebenszeit hinaus gültigen, Bezeichner referenziert, der aus dem Paar zweier Socket-IDs besteht.

Mithilfe dieser IDs können Kontrollthreads veränderte Server-Socket IP-Adressen vom GKM erkannt und für den Verbindungswiederaufbau bezogen werden.

6.4.5 Gemeinsam genutzte Sockets

Gridkanalsicherung wird von Eigenschaften eines zugrundeliegenden Betriebssystems beeinflusst. In Linux kann ein Socket von mehreren Threads eines oder mehrerer Prozesse verwendet werden. Das heißt insbesondere, dass ein, vor dem Checkpoint von mehreren Threads gemeinsam genutzter, Socket beim Restart ausschließlich durch *einen* Thread wiederhergestellt werden darf. Eine Inkonsistenz tritt auf, wenn jeder Prozessthread ein vor Checkpointing existierendes Socket wiederherstellt.

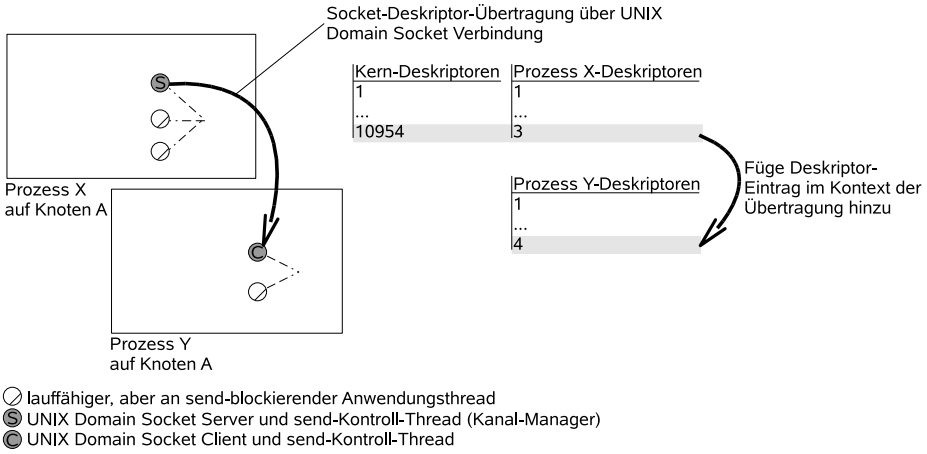

Abbildung 6.2: UNIX Descriptor Passing

Der unter Kapitel 6.4.3 erwähnte Kanal-Manager stellt exklusiv gemeinsam genutzte Sockets im Post-Checkpoint- beziehungsweise Restart-Callback eines Prozesses in einer Mehrprozessanwendung wieder her. Unter Verwendung des *UNIX descriptor passing* Mechanismus, siehe [133], erhalten die verbleibenden Prozesse Zugriff auf wiederhergestellte Sockets. Hierbei wird der wiederhergestellte Socket-Deskriptor vom Kanal-Manager zu anderen Kontrollthreads über eine UNIX Domain Socket Verbindung gesendet. Diese wird über einen vom Kanal-Manager vergebenen eindeutigen Schüssel, welcher mit einem Kanal assoziiert ist, initiiert. Bei der Übertragung des Deskriptors wird inhärent ein entsprechender Eintrag in der Deskriptor-Tabelle des Empfänger-Prozesses im Kern erzeugt, welcher auf den, bereits im Kern erzeugten, Socket verweist.

Dieser Mechanismus garantiert korrekte Socketwiederherstellung und besitzt den Vorteil, Kernel- als auch Bibliothekscheckpointer zu unterstützen. Einerseits werden Mehrfach-Rekonstruktionen eines Sockets vermieden. Andererseits müssen gemeinsam genutzte Sockets nicht anhand von *fork* und inhärenter Vererbung aller offenen Deskriptoren an Kind-Prozesse wiederhergestellt werden, wie es bei Bibliothekscheckpointern üblich ist.

Socket-Deskriptoren werden während der normalen Programm-Laufzeit in einer aufsteigenden Nummerierung vergeben. Wird ein mittiger Deskriptor geschlossen, entsteht eine Lücke zwischen den DeskriptorIDs. Hingegen werden DeskriptorIDs bei der Socket-Wiederherstellung in aufsteigender Weise vergeben, ohne Lücken. Daher muss eine Socket-ID mit der initialen DeskriptorID zum Checkpointzeitpunkt gespeichert und beim Restartzeitpunkt berücksichtigt werden. Dies führt unter Umständen zur Anpassung von DeskriptorIDs, beispielsweise mithilfe des Systemaufrufs *int dup2(int fildes, int fildes2)*. Bei *dup2* wird ein Quelldeskriptor dupliziert und geschlossen. Der neue Deskriptor verweist auf den Quelldeskriptor, besitzt jedoch die beim Aufruf übergebene Kennung.

.

6.5 GKS Phasen

Das GKS-Protokoll gliedert sich in drei Phasen und wird anhand von Callbacks zum Pre- und Post-Checkpoint- als auch Restart-Zeitpunkt ausgeführt.

6.5.1 Pre-Checkpoint-Phase

6.5.1.1 Blockierung der Erzeugung weiterer Känale

Zwischen dem Pre-Checkpoint-Callback und der Prozess-Synchronisierung existiert ein kurzes Zeitintervall, in welcher potentiell neue Kanäle erzeugt werden können. Dies muss verhindert werden, da nach dem Ende der Pre-Checkpoint-Callback-Phase kein Kanal mehr geleert werden kann. Neue Kanäle werden nicht erzeugt, wenn *accept* und *connect* aufrufende Threads von der GKS-Bibliothek bis zum Ende des Checkpoints schlafen gelegt werden.

6.5.1.2 Bestimmung eines Kanal-Managers

Damit ein Kanal in absehbarer Zeit geleert werden kann, muss verhindert werden, dass fortwährend Nachrichten über einen zuverlässigen Kanal gesendet werden. Hierbei müssen zwei Aspekte beachtet werden. Einerseits kann ein Socket von mehreren Threads eines oder mehrerer Prozesse referenziert werden, siehe Kapitel 6.4.5. Somit kann jeder dieser Threads einen Marker senden, beziehungsweise empfangen. Die dafür notwendige Protokollimplementierung ist komplex und kann zu Effizienzeinbußen führen, weil abgesichert werden muss, dass von n Threads gesendete Marker auch bei n Threads angekommen sind. Andererseits können Anwendungsthreads insbesondere bei *recv* blockieren, sodass das Protokoll nicht durch diese Threads ausgeführt werden kann.

Der erste Aspekt wird gelöst, indem gemeinsam genutzt Sockets exklusiv vom Kanal-Manager, siehe Kapitel 6.4.3, überwacht werden. Jeder send- und recv-Kontrollthread sendet eine Kanal-Manager-Anfrage an das lokale Socket-Management der GKS-Bibliothek. Das heißt, in einer Zweiprozessanwendung, in welcher dem Kindprozess die Socketdeskriptoren des Elternprozesses vererbt wurden, werden zwei Anfragen gesendet. Das lokale Socket-Management bestimmt einen Kontrollthread als Kanal-Manager und informiert alle anderen darüber.

Der zweite Aspekt wird gelöst, indem die GKS-Ausführung von Anwendungsthreads auf Kontrollthreads verschoben wird. Send/recv-Kontrollthreads senden das SIGALRM Signal ausschließlich an Anwendungsthreads, die derzeit an *recv* oder *send* blockieren, um sie in der zugehörigen Unterbrechungsroutine temporär schlafen zu legen. Weil Kontroll- und Anwendungsthreads zuvor als solche markiert worden sind, wird die Unterbrechung nur

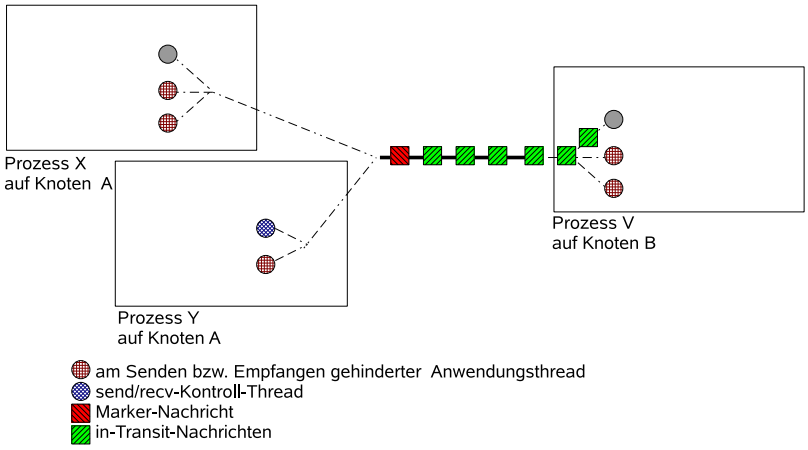

Abbildung 6.3: Kanalleerung mit Markernachricht bei gemeinsam genutzten Sockets

für Anwendungsthreads erzeugt. Von nun an kann kein Anwendungsthread bis zum Ende der Post-Checkpoint-Phase Nachrichten versenden oder empfangen.

6.5.1.3 Kanalleerung

Soll ein Kanal geleert werden, müssen sich die involvierten und voneinander entfernten *send*- und *recv*-Kontrollerthreads koordinieren, damit der *recv*-Kontrollthread an dem Socket lauscht, auf dem der, vom *send*-Kontrollthread gesendete, Marker eintrifft. Hierbei muss der Grid Kanal Manager einbezogen werden. Nachrichten, die vor dem Marker eintreffen können, werden vom Marker getrennt und in einem Kanal-bezogenen Kanalpuffer abgelegt.

Im Normalfall wartet nur ein Anwendungsthread pro einem Kanal, es können jedoch auch mehr sein. Der Betriebssystem-Scheduler entscheidet an diesem Punkt nicht-deterministisch über die Nachrichtenzuweisung an einen Thread. Daher wird der Kanalpuffer mit dem Anwendungsthread assoziiert, welcher als letztes an dem Kanal gelauscht hat. Dies entspricht einem möglichen Zustand während der fehlerfreien Ausführung.

Ein Marker kann auf verschiedene Weise realisiert werden. Da die zugrundeliegende TCP-Schicht Anwendungsdaten fragmentiert, kann ein Marker, je nach Größe, über mehrere TCP-Pakete verteilt sein. Daher müssen die empfangenen Daten byteweise mit dem Marker abgeglichen werden, um zu erkennen, ob der Marker empfangen wurde.

Analog kann jeder zu versendende Nachrichtenpuffer der Anwendung mit einem speziellen GKS-Header versehen werden. Ein Header-Bit indiziert, ob der Puffer dem Marker entspricht oder nicht. Anstelle byteweise abzugleichen, ist eine Überprüfung auf Pufferebene ausreichend. Dieser Ansatz führt jedoch zu Leistungseinbußen, wenn ein Header vorangestellt, überprüft und entfernt werden muss.

Alternativ kann der *shutdown*-Befehl verwendet werden, um das Senden eines FIN Pakets auf TCP-Ebene auszulösen, welches vom Kern als solches erkannt wird.

6.5.1.4 Socketabbau

Checkpointer, die Sockets nicht sichern und wiederherstellen können, werden vom Protokoll unterstützt, indem Sockets vor der Sicherung geschlossen und danach wiederaufgebaut werden. Dies geschieht transparent für Anwendungsthreads, da sie nicht an den nativen *send*- und *recv*-Routinen, sondern an den Wrappern der GKS-Bibliothek blockieren.

6.5.2 Post-Checkpoint-Phase

Nach einer verteilten Sicherung können kanalerzeugende Aufrufe deblockiert werden, da alle relevanten Kanäle geleert wurden. Ruft ein Anwendungsthread *connect* oder *accept*, wird er in der zugehörigen Wrapper-Funktion nicht mehr schlafen gelegt. Die vor Checkpointing schlafen gelegten Threads werden deblockiert.
Sockets müssen nur dann wiederhergestellt werden, wenn sie in der Pre-Checkpoint-Phase geschlossen wurden. Rücksicht auf veränderte Server-Socket-Adressen muss nicht genommen werden, da keine Migration stattfand. Die Wiederherstellung von Sockets wurde bereits in Kapitel 6.4.5 dargestellt.

Anwendungsthreads, welche vor dem Checkpointing vom Senden und Empfangen abgehalten wurden, werden aufgeweckt. Vor der Verarbeitung neuer empfangener Nachrichten müssen die alten zwischengespeicherten Nachrichten von den Anwendungsthreads konsumiert werden.

6.5.3 Restart-Phase im Migrations-Kontext

Die Restart-Phase nach der Migration eines Prozesses ähnelt dessen Post-Checkpoint-Phase. Wurde zwischenzeitlich ein Server-Socket migriert, unterscheidet sie sich zur Post-Checkpoint-Phase im Ausführungsort.

Kanalerzeugende Aufrufe werden, wie unter Kapitel 6.5.2 beschrieben, deblockiert.

Anschließend werden (gemeinsam genutzte) Sockets wiederhergestellt. In diesem Zusammenhang ist zu erwähnen, dass Checkpointer mit unterschiedlichen Fähigkeiten existieren, wie

- dem Sichern und Rekonstruieren von Sockets mit Berücksichtigung von Server-Socket-Migrationen (Typ 1),
- dem Sichern und Rekonstruieren von Sockets, jedoch ohne Migrations-Unterstützung (Typ 2),

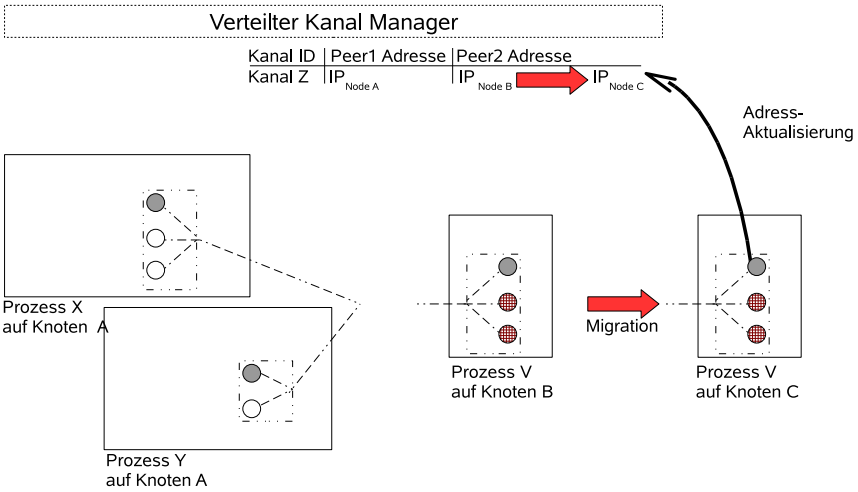

Abbildung 6.4: IP-Adress-Aktualisierung bei Migration

- ohne die Fähigkeit, Sockets sichern und wiederherstellen zu können (Typ 3).

Das GKS muss Checkpointer des Typs 2 und 3 einerseits um Socketsicherung und -wiederherstellung, siehe Kapitel 6.4.5, komplementieren. Andererseits muss GKS Migration gewährleisten. Hierbei ermittelt der zuständige Kontrollthread eines Clientsockets die aktuelle Serversocket IP-Adresse, die zuvor vom gegenüberliegenden Kontrollthread beim Grid Kanal Manager aktualisiert wurde, siehe Abbildung 6.4. Zusätzlich muss der Clientsocket Kontrollthread den Kontrollthread des Serversockets kontaktieren, um eine Verbindungswiederherstellung erst dann zu initiieren, wenn der Serversocket an der neuen Adresse auf eingehende Verbindungen wartet.

Anwendungsthreads, welche vor dem Checkpointing vom Senden und Empfangen abgehalten wurden, werden aufgeweckt. Bevor neu eintreffende Nachrichten verarbeitet werden können, müssen zwischengespeicherte Nachrichten von den Anwendungsthreads konsumiert werden.

6.6 GKS und verschiedene Ein-/Ausgabe-Modelle

Es existieren verschiedene Ein-/Ausgabemodelle, welche von der Kanalzustandssicherung unterstützt werden müssen.

Bei *blockierender Ein-/Ausgabe* kehrt beispielsweise der Systemaufruf *recv* erst dann zurück, wenn die Daten im Kern angekommen und in den Anwendungspuffer kopiert worden sind. Bei *nicht-blockierender Ein-/Ausgabe* überprüft die Anwendung *kontinuierlich*, ob eine Ein-/Ausgabeoperation terminiert hat, wobei Prozessorzeit ineffizient verbraucht wird. Beide Modelle werden von beschriebenen GKS, ohne zusätzliche Erweiterungsmassnahmen, un-

terstützt.

Ein-/Ausgabe-Multiplexing wird mithilfe des *select* Aufrufs realisiert. Eine Ein-Thread-Anwendung beauftragt den Kern, sie darüber zu informieren, dass sie für eine Ein-/Ausgabe-Operation bereit ist. Im Vergleich zu blockierender Ein-/Ausgabe kann die Ein-Thread-Anwendung *an mehreren Deskriptoren gleichzeitig* lauschen, zu denen umgeschalten wird, wenn an ihnen Daten oder Speicher vorliegen. Im Vergleich zu Multithreading mit blockierender Ein-/Ausgabe, was eng verwandt mit Ein-/Ausgabe-Multiplexing ist, kann dieses Modell mit ausschließlich einem Thread implementiert werden. Im Hinblick auf GKS ist zu beachten, dass die Kanalkontrolle an den Kontrollthread übergeben werden muss. Dies bedingt, dass der Kontrollthread für die GKS-Ausführungsdauer, vom Kern über anstehende Ein-/Ausgabe-Operationen informiert werden muss, anstelle des Anwendungsthreads, ansonsten wird der Marker nicht erkannt. Hierzu muss das Deskriptorset des Anwendungsthreads dem Kontrollthread temporär und exklusiv zugeordnet werden.

Beim *signalbasierten Ein-/Ausgabemodell* wird ein Handler für das SIGIO-Signal installiert, der ausgeführt wird, wenn eine Ein-/Ausgabeoperation beginnt. Der Anwendungsprozess wird nicht blockiert.

Asynchrone Ein-/Ausgabe hingegen bedeutet, dass die Anwendung dem Kern mitteilt, eine Operation zu starten. Im Gegensatz zu signalbasierter Ein-/Ausgabe benachrichtigt der Kern die Anwendung anschließend nicht darüber, wann die Operation *initiiert* werden kann, jedoch wann sie *beendet* wurde. Hierbei kehrt der Aufruf unmittelbar zurück, sodass der Prozess für die Dauer der Ein-/Ausgabeoperation nicht blockiert wird. Das Kernsignal wird erst dann an den Prozess gesendet, wenn die Daten in den Anwendungspuffer kopiert worden sind.

GKS kann erweitert werden, um signalbasierte und asynchrone Ein-/Ausgabe zu ermöglichen. Damit der Kontrollthread, von anderen Anwendungsthreads ungestört, über eingehende (Marker-) Nachrichten benachrichtigt wird und dadurch den Kanal sichern kann, muss das Signal an den Kontrollthread umgeleitet werden. Hierzu installiert der Kontrollthread beispielsweise einen Handler für das SIGIO-Signal. Der auf dem Kanal arbeitende Anwendungsthreads maskiert dies temporär. Zwischen Handlerregistrierung und Signalmaskierung muss sichergestellt werden, dass kein Signal verloren geht.

6.7 Nebenläufiges Kanal-Management

Bei nebenläufigem Kanalmanagement (NKM) werden Netzwerkaktivitäten nicht temporär stillgelegt, Prozesse werden nicht synchronisiert. Im Gegensatz zu GKS und koordiniertem Checkpointing wird NKM parallel zur Anwendung ausgeführt.

Abhängig von der Modellierung eines verteilten Systems ergeben sich unterschiedliche Konzeptionen des NKMs. Wird ein verteiltes System als Graph modelliert, in dem jeder Knoten einem Prozess und eine Kante einem FIFO Kanal entspricht, besitzt jeder Knoten minde-

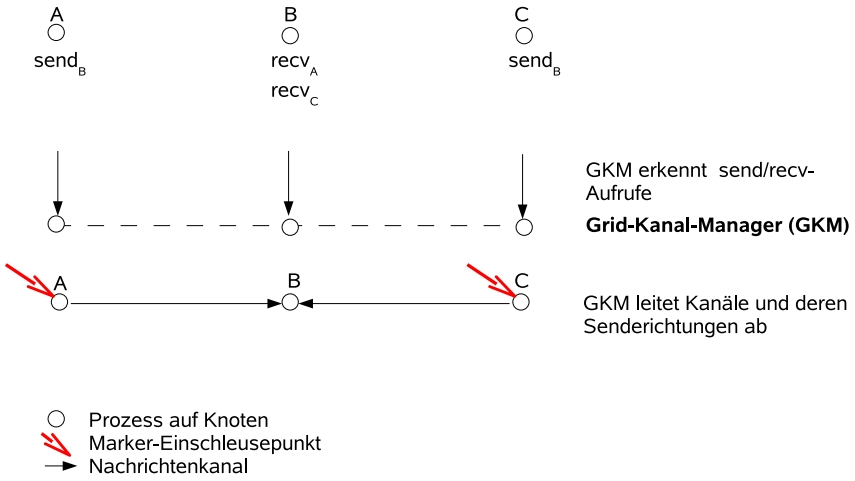

Abbildung 6.5: Bestimmung des Marker-Einschleusepunktes

stens eine Verbindung zu einem anderen Knoten.

Bei NKM nach [29] bedingt die Marker-initiierte Checkpoint-Erzeugung, dass jeder Prozess über die zugrundeliegende Kommunikationsinfrastruktur *erreichbar* ist. In diesem Zusammenhang sind sequentielle Prozesse und der Einbezug uni- und bi-direktionaler Kanäle zu berücksichtigen.

Ein sequentieller Prozess nach [75] wird entweder *ausgeführt*, *wartet* oder hat *terminiert*. Beispielsweise gilt ein wartender Prozess P als *blockiert*, wenn er Prozess Q Nachrichten senden will, Q auf dessen Empfang jedoch nicht wartet. Stellt der initiale Empfang der Marker-Nachricht den Sicherungsimpuls dar und kann der Marker jedoch aufgrund einer *Blockierung* nicht zum Zielknoten ausgeliefert werden, kann der globale Schnappschuss nicht unmittelbar erstellt werden.

Letzteres tritt ebenfalls auf, wenn Knoten B nur über einen *ausgehenden* Kanal mit Knoten A verbunden ist, A den Marker besitzt, ihn jedoch darüber nicht an B aufgrund der einseitigen Sende-Richtung weiterleiten kann[3].

Um alle Prozesse zu erreichen und eine Sicherung auszulösen, müssen Marker an mehreren Stellen eingeleitet werden. Die Herausforderung besteht darin, jene Kanalenden zu ermitteln, über welche Marker eingeleitet werden müssen, damit ein globaler Schnappschuss in endlicher Zeit erzeugt werden kann. Mit Blick auf die GCA und GKS-Komponenten muss das Wissen des Job-Managers (JM), siehe Kapitel 3.2.1, des Job-Einheit-Managers (JEM), siehe Kapitel 3.2.2, und des Grid-Kanal-Managers (GKM), siehe Kapitel 6.4.4, ein-

[3] [29] ignoriert diese realen Szenarien, indem bidirektionale Kanäle und der Ausschluss blockierender Prozesse vorausgesetzt werden, beziehungsweise dass jeder Prozess absichert, dass kein Marker für immer in einem eingehenden Kanal bleibt und *zweitens* der Prozesszustand in endlicher Zeit nach der Algorithmusinitiierung, genommen wird.

bezogen werden. Der JM und die involvierten JEMs bestimmen alle zum Job gehörenden Prozesse. Der GKM ermittelt alle Kanäle, die von den Job-Einheits-Prozessen erzeugt werden (überwache *connect*- und *accept*-Aufrufe) und identifiziert deren Senderichtungen (überwache *send*- und *recv*-Aufrufe), siehe Abbildung 6.5. Diese Informationen werden verwendet, um einen gerichteten Graph zu erzeugen, bei dem eine Kante einem Kanal und ein Knoten einem sendenden und/oder empfangenden Prozess entsprechen. Die Marker-Sende und -Empfangsregeln nach [29] werden simuliert, um zu überprüfen, ob mit einer initialen Menge von Einschleusepunkten alle Kanäle geleert werden können. Trifft dies nicht zu, muss die Menge abgeändert werden.

Der JM muss erkennen, dass der Algorithmus terminiert hat, um überlappende Checkpoint-Aktionen zu vermeiden. Der JC erkennt die Terminierung, nachdem für alle Job-Einheiten entsprechende Abbilder im verteilten Dateisystem abgelegt worden sind. Um alle Zustände einzusammeln, bedarf es daher keiner vollvermaschten Topologie, damit jeder Prozess das eigene Abbild allen anderen zusenden kann. Letzteres ist nicht möglich, da es zu erhöhter Netzwerkbelastung und lokaler Speicherbelastung bei zunehmender Verteilung und Adressraumgröße führt.

Die Lösung des Erreichbarkeitsproblems im Bezug auf *blockierende Prozesse* ist komplizierter. Um zu wissen, ob ein Sender einem Empfänger eine Nachricht senden kann, muss der Wartestatus zwischen Nachbarprozessen kommuniziert werden. In [104] führt ein Signal-Schema, dass oberhalb einer verteilten Berechnung ausgeführt wird, auch bei blockierenden Prozesse dazu, dass der Berechnungsinitiator bezüglich einer Berechnungsterminierung benachrichtigt werden kann.

Bei gemeinsamer Socketnutzung durch alle Prozessthreads wird die Protokoll-Implementierung vereinfacht, wenn nur ein Prozessthread den Marker sendet, um den Checkpoint auszulösen und/oder das Kanal-Flag zu setzen. Die Kanal-Manager-Semantik, siehe Kapitel 6.4.3, muss für den konsistenten Wiederaufbau gemeinsam genutzter Sockets angewendet werden.

6.8 Verwandte Arbeiten

DMTCP [15], [123] realisiert anwendungstransparentes, koordiniertes Checkpointing/Restart einer im Cluster verteilten Anwendung, hierbei wird ausschließlich ein Checkpointer-Paket einbezogen. Migrierende Sender- und Empfängerprozesse werden unterstützt. Library Interposition wird verwendet, unter anderem um Sockets kontrollieren zu können. Ein Kanal wird mit den IDs zweier Sockets assoziiert und bei einem clusterweiten Verzeichnisdienst hinterlegt. Gemeinsam genutzte Sockets werden exklusiv mithilfe der sogenannten Election Leader (EL) Abstraktion verwaltet. Ein EL schiebt in-Transit Nachrichten aus einem Kanal heraus, indem ein spezieller Marker gesendet wird. In-Transit Nachrichten werden als Teil des Empfängerprozessadressraums gesichert. Nach der Prozess-Sicherung werden herausgeschobene Kanalinhalte von der Empfängerseite zurück zur Senderseite gesendet. Der Senderseite überträgt die Daten erneut zum Empfänger, um dessen Kernelpuffer zu befüllen, bevor die Anwendungsausführung fortgeführt wird.

Ein separater Restartprozess stellt zunächst Deskriptoren regulärer Dateien, lauschender Sockets und nicht initialisierter Sockets pro Knoten wieder her, welche den im Anschluss erzeugten Kindprozessen vererbt werden. Socket-zugehörige Kanaleinträge des Verzeichnisdienstes werden hinsichtlich aktueller Adress- und Port-Daten angepasst, um Verbindungswiederherstellungen nach Migrationen zu ermöglichen. Bevor die Anwendung weiter rechnet, werden die in-Transit-Daten erneut vom Sender zum Empfänger übertragen, um dessen Kernel-Socket-Puffer zu füllen.

DMTCP und GKS unterstützen beide gemeinsam genutzte Sockets und fangen Bibliotheksaufrufe mithilfe der Library Interposition-Technik ab. GKS vererbt jedoch gemeinsam genutzte Sockets nicht im Benutzeradressraum von einem Wurzelprozess an einen Kindprozess. Damit ist die GKS flexibler, weil sie für Kernel-, als auch Bibliotheks-Checkpointer geeignet ist. Gesicherte in-Transit-Nachrichten werden bei GKS einmalig und unmittelbar beim Post-Checkpoint- beziehungsweise Restart-Zeitpunkt verarbeitet, während DMTCP in-Transit-Nachrichten nach dem Checkpoint zum Sender zurückschickt, der sie anschließend erneut zum Empfänger sendet. Analog werden sie beim Restart zum Empfänger gesendet. Die Architektur integriert zudem keine heterogenen Checkpointer.

DCR [96] stellt ein anwendungstransparentes Checkpoint/Restart-Framework für verteilte Systeme dar, welches auf modifiziertem BLCR und TCPCP2 [9] basiert. DCR verwendet sogenannte Filter damit auf TCP-Kanälen keine Nachrichten vor dem Sichern ausgeliefert oder gesendet werden können. *Einzelne* Prozesse können bereits nach Socket-Sicherung Nachrichten im Kernel TCP Stack empfangen, anstelle darauf warten zu müssen, bis alle Prozesse gesichert wurden. Die lokal zurückgehaltenen Nachrichten werden erst dann ausgeliefert, wenn die Filter zurückgesetzt wurden. Zu sendende Nachrichten werden solange gehindert, an den entfernten Prozess ausgeliefert zu werden, bis der Empfängerprozess zu Ende gesichert wurde. Da ausschließlich Kommunikations-Operationen, jedoch keine anderen Anwendungsberechnungen blockiert werden, wird dieses Verfahren von den Autoren als *on-demand-blocking* bezeichnet.

Sockets werden mithilfe TCPs inhärenter Sessionkonservierung (engl. TCP session preservation) migriert. Hierzu werden aktive TCP Verbindungen pausiert. TCP startet einen Timeout-Timer, um potentiell verloren gehende Segmente, nach Timerablauf erneut zu übertragen. Es erfolgt damit keine Kanalleerung mithilfe eines Markers.

Sockets werden wiederhergestellt anhand von Socketinformationen, die mit dem TCPCP2 Werkzeug gesichert wurden. Der im Checkpoint gesicherte TCP-Zeitstempel wird an den, derzeit auf dem Zielknoten gültigen Zeitstempel, angepasst. Damit wird die als Protect Against Wrapped Sequence Numbers (PAWS) [79] bekannte TCP-Erweiterung eingehalten. Demnach kann ein Segment als veraltetes Duplikat erkannt werden, falls dessen Zeitstempel kleiner ist als jener eines kürzlich zuvor auf diesem Kanal empfangenen Segments. Darauf aufbauend kann das veraltete Segment ignoriert werden. Dieser Algorithmus erfordert keine Uhrensynchronisation zwischen neuem Sender und Empfänger, da der Zeitstempel eine monoton wachsende Seriennummer darstellt.

Bei DCR muss das zugrundeliegende Betriebssystem modifiziert werden, um TCPCP2-Funktionalität nutzen zu können. Aus dem Ansatz geht nicht hervor, wie das eng an TCP-Eigenschaften angelehnte Kanalmanagement Kanäle konsistent wiederherstellt, wenn zwischen Fehler und Wiederherstellung mehr Zeit vergeht, als der TCP Timeout-Timer aktiv ist. Darüber hinaus ist unklar, wie ein Kanal konsistent wiederhergestellt wird, wenn Client und Server gleichzeitig migriert werden, beziehungsweise wenn ein Prozess Client und Server (Duplex-Modus) in einem ist.

In LAM MPI [125] werden pro MPI-Kanalende die Anzahl gesendeter und empfangener Bytes, sogenannte Bookmarks, gespeichert, wobei in-Transit Nachrichten vorliegen, wenn mehr Bytes gesendet als auf Gegenseite empfangen wurden. Empfangene Bytes werden über *out-of-band* Signalisierungs- und Kommunikationsdienste ermittelt, die von der LAM-Umgebung bereitgestellt, jedoch nicht mitgesichert werden. Ein P2P-Kommunikationsmodul wird aufgefordert, im Netzwerk befindliche Nachrichten zu verarbeiten, bis die Anzahl gesendeter und empfangener Bytes übereinstimmt.

Tritt ein *MPI receive vor* einer Checkpoint-Anfrage auf, erreicht die Nachricht die Gegenseite jedoch erst nachdem die Kanalleerung begonnen hat. Hierbei besteht die Gefahr, zu wenig Nachrichten aus dem Kanal zu schieben. MPI garantiert, dass diese unerwarteten Nachrichten gepuffert werden und nicht verloren gehen. *Während der Kanal geleert wird,* werden MPI Prozesse daran gehindert, MPI-Bibliotheksaufrufen auszuführen. Blockiert ein aktueller MPI-Aufruf eines Anwendungsthreads, wird eine Unterbrechung erzeugt, welche die Freigabe des MPI-Aufrufs erwirkt und Kontrolle an den Checkpoint-Thread übergibt.

LAM MPI nutzt BLCR als zugrundeliegenden Checkpointer und dessen Callback-Infrastruktur. Ein Thread-basierter Callback propagiert den Checkpoint-Aufruf zum MPI-Prozess, blockiert MPI-Aufrufe und initiiert den Bookmark-Austausch und leert den MPI-Kanal. Beim Restart erzeugt ein Signal-basierter Callback den neuen *mpirun*-Prozess. Der Thread-basierte Callback-Thread des wiederhergestellten MPI-Prozesses erzeugt einen neuen Socket pro MPI-Kanalende und deblockiert MPI Bibliotheks-Aufrufe.

Ähnlich wie im GKS-Protokoll bilden Callbacks die Plattform zur Ausführung des Kanal-Managements. LAM MPI unterstützt jedoch kein Kanal-Management für heterogene Checkpointer. Zudem wird keine Lösung für gemeinsam genutzte Sockets aufgezeigt. Dies liegt jedoch darin begründet, dass einerseits Prozesshierarchien nach [53] in MPI nicht existieren und andererseits mehrere Threads pro MPI-Prozess nicht unterstützt werden müssen von einer MPI-Implementierung.

Virtual Network Address Translation (VNAT) [137] beschreibt Anwendungsmigration mit aktiven Netzwerkverbindungen. Bezeichner von Kanalenden werden durch virtuelle Adressen ersetzt und von physikalischen Netzwerkadressen unterhalb des Transport-Protokolls entkoppelt. VNAT sichert den Verbindungszustand des zu migrierenden Endpunkts und legt ihn still. Der Verbindungszustand des nicht migrierenden Endpunkts wird im Transport-Protokoll und in der Anwendung mithilfe eines sogenannten Verbindungsmigrations-Helfers am Leben gehalten, indem unter anderem ein anwendungsunabhängiger Helfer den TCP keep-alive-Timer des Kanal-Peers deaktiviert. Anwendungsgebundene Helfer können integriert werden, um Anwendungs-Timer zu kontrollieren. Der Checkpointer verhindert, dass Nachrichten während des Checkpointings ausgeliefert und versendet werden können, anhand gesetzter Spin-Locks. Die Verbindungswiederaufnahme bedingt, die Abbildung virtueller auf physikalische Endpunkt-Bezeichner zu aktualisieren. Die VNAT-Instanz des Migrations-Zielrechners benachrichtigt jene des nicht migrierten Anwendungsteils. VNAT aktiviert die migrierte Verbindung unter anderem, indem die, vor der Migration gültigen, TCP-Timerzustände wiederhergestellt werden.

Die Integration virtueller Endpunktbezeichner ermöglicht es, Änderungen physikalischer Netzwerkadressen vor der Anwendung zu verbergen. Wie bei DCR wird eine explizite Kanalleerung vermieden, jedoch führt die TCP-inhärente Paketneuübertragung dazu, dass keine Nachrichten verloren gehen. Die Modifizierung des TCP-Protokolls beinhaltet zwei Betriebssystemschnittstellen, damit zwei Sequenznummern angepasst werden können. Aus dem Ansatz geht nicht hervor, wie beide Kanalenden, beziehungsweise Prozesse im Duplex Modus migriert werden können. VNAT wurde im Kontext des *zap* Checkpointers eingesetzt. Kanal-Management mit heterogenen Checkpointern ist jedoch nicht vorgesehen.

In [29] wird die nebenläufige Berechnung des globalen Zustands eines verteilten Systems beschrieben. Ein oder mehrere Prozesse initiieren eine Checkpoint-Welle, indem sie eine eigene Sicherung vornehmen und eine Markernachricht über jeden ausgehenden FIFO-Kanal schicken, bevor neue Anwendungsnachrichten darüber ausgesendet werden. Bei initialem Markerempfang wird der Empfänger-Prozess gesichert. Die vom Marker herausgeschobenen Nachrichten werden in diesen Checkpoint integriert, verlorene Nachrichten werden hierdurch verhindert. Ein Marker wird über jeden ausgehenden Kanal gesendet, bevor neue Anwendungsnachrichten vom Prozess versandt werden. Wird ein Marker zum wiederholten Mal empfangen, seit der Prozesssicherung, werden alle auf diesem Kanal eingetroffenen Nachrichten mit diesem Checkpoint assoziiert, verwaiste Nachrichten werden hierdurch verhindert. Der Algorithmus terminiert, insofern kein Marker für immer in einem eingehenden Kanal bleibt und Prozesssicherungen erfolgreich abgeschlossen werden.

Dieser Ansatz ermöglicht eine globale Zustandsermittlung für Anwendungen mit einer spezifischen Kommunikationstopologie. In der Praxis ist nicht jeder Anwendungsprozess über einen Kommunikationskanal, von der verbleibenden Anwendung aus, erreichbar. Fehlt ein Kanal, bleibt die Checkpoint-Aufforderung aus, weil der Marker nicht propagiert werden kann. Da Prozess- und Kanalzustände über Anwendungskanäle eingesammelt werden, kann einerseits erheblicher Aufwand, andererseits Inkonsistenz entstehen, wenn mindestens ein Rückkanal fehlt. Der Ansatz berücksichtigt zudem keine Linux-typischen Gegebenheiten, wie gemeinsam genutzte Sockets.

6.9 Zusammenfassung

Zustände von Kommunikationskanälen sind Teil des globalen Zustands einer verteilten Anwendung. Werden in-Transit Nachrichten beim Checkpointing ignoriert, kann es zu verlorenen Nachrichten bei Abbildern koordinierten Checkpointings sowie zu verlorenen und verwaisten Nachrichten bei Abbildern nebenläufigen Checkpointings kommen, sodass Inkonsistenzen beim Restart entstehen.

In diesem Kapitel wurde eine neue Architektur vorgestellt, anhand derer Kanäle von heterogenen Checkpointern konsistent gesichert und wiederhergestellt werden können. Eine verteilte Anwendung und zugrundeliegende Checkpointer-Pakete müssen nicht modifiziert werden, weil die GKS oberhalb derer, in diversen Callbacks, ausgeführt wird, die mithilfe des Library-Interposition Mechanismus transparent in die Anwendung eingeblendet werden.

Mit der vorgestellten Lösung werden erstmals heterogene Checkpointer integriert, die gemeinsam Kanäle sichern und wiederherstellen. GKS integriert insbesondere Checkpointer, welche keine Socket-Unterstützung anbieten.

Im Allgemeinen können Sockets gemeinschaftlich von mehreren Prozessen und mehreren Threads eines Prozesses genutzt werden. Mit GKS können diese Sockets erstmalig mit Bibliotheks- und Kernel Checkpointer gleichzeitig gesichert und wiederhergestellt werden. Dies wurde unter anderem mithilfe des UNIX Descriptor-Passing-Mechanismus erreicht.

Im Gegensatz zu Ansätzen, die auf TCP-Sessionkonservierung basieren, kann bei GKS ein beliebig großes Zeitinterall zwischen einem Anwendungsfehler und der Wiederherstellung liegen, weil GKS nicht an TCP-Timervorgaben gebunden ist.

Weiterhin ermöglicht GKS, dass Client und Serversocket, mit einer kurzen Berechnungsunterbrechung, *gleichzeitig* migriert werden können. Dies wird bei den beschriebenen Live-Migrationsansätzen nicht gewährleistet.

Zudem erlauben GKS-interne Strukturen, dass ein Prozess Client- und Serversocket zur selben Zeit ausführen kann, ohne dass es beim Checkpointing/Restart zu Verklemmungen kommt.

Im Gegensatz zu DMTCP werden in-Transit-Nachrichten effizient verarbeitet, gleichzeitig aber auch interaktive Anwendungen unterstützt, da der im Kern befindliche TCP-Puffer, für ausserhalb der Sendereihenfolge empfangene Pakete (engl. out-of-order), nach Erhalt des Markers auf Anwendungsebene, geleert worden ist.

7 Adaptives Grid-Checkpointing

Einerseits sind Grid-Ressourcen dynamisch verfügbar, andererseits ist die *Ressourcennachfrage* einer Grid-Anwendung, aus der Sicht eines Grid-Computing Systems, im Vorfeld nicht immer einschätzbar. Um die Performanz des Gesamtsystems und der einzelnen Job-Anwendungen zu erhöhen, als auch Knoten mit verschiedenen Fehlerwahrscheinlichkeiten zu unterstützen, bedarf es adaptiver Checkpointingstrategien.

7.1 Hintergrund und Motivation

Idealerweise schützt Grid-Checkpointing in heterogenen Umgebungen davor, dass wichtige Zwischenzustände von Berechnungen infolge von Hard- und Software-Fehlern verloren gehen, ohne dass die System- und Anwendungsausführung gravierend beeinträchtigt wird. Ein System, welches all diesen Anforderungen gerecht wird, existiert bis dato nicht. Die Ursache hierfür liegt in der *Komplexität der zu betrachtenden Einzelfälle, beziehungsweise der Anzahl Checkpoint-relevanter Parameter.*

Obwohl Checkpointing seit Jahrzehnten wissenschaftlich in Theorie und Praxis erforscht wird, existiert beispielsweise kein Checkpointingprotokoll, welches sich *optimal* hinsichtlich *aller* folgenden Aspekte *zugleich* verhält: Performanz, Ausgabelatenz[1], Speicheraufwand, Abbildverwaltung, Domino-Effekt, Waisenprozesse und Aufwand beim Zurückrollen auf ältere Zustände [45]. Zudem sind die meisten der verteilten Checkpointingprotokolle für Cluster-Umgebungen ausgelegt. Grid-spezifische Charakteristiken, wie Ressourcen-Dynamik, Heterogenität und variable/hohe Netzwerklatenzen, werden bisher nur ansatzweise berücksichtigt.

Das Ziel adaptiven Checkpointings ist es, Fehlertoleranz-Effizienz zu erhöhen, ohne die Konsistenz zu beeinträchtigen. Allerdings existiert keine zentrale Gridinstanz, die eine, für alle Anwendungen gültige Checkpointingstrategie bestimmt. Adaptives Checkpointing muss dynamische Ausführungskontexte und Fehlerverhalten erkennen und bei der Strategiebestimmung berücksichtigen.[2] Der selbstregulierende Charakter des adaptiven Checkpointings führt zu einem fortlaufendem Abgleich des aktuellen Checkpointingverhaltens

[1] Werden Determinanten und Abhängigkeitsinformationen im fehlerfreien Betrieb aufgezeichnet, entsteht eine Verzögerung.

[2] Ein Ausführungskontext setzt sich in dieser Betrachtung aus dem Verhalten des Systems und dem Verhalten der darauf ausgeführten Anwendung zusammen.

mit dem aktuellen Ausführungskontext. Werden zuvor definierte Kriterien erfüllt, wird das Checkpointingverhalten einer Anwendung angepasst.

7.2 Individuelle Ausführungskontexte

Eine Checkpointingaktion beeinflusst die Systemausführung. Ressourcen wie Prozessorzyklen, Netzwerkbandbreite, Speichermedien werden von Checkpointingprotokollen in unterschiedlichem Ausmaß in Anspruch genommen. Während bei unkoordiniertem Checkpointing viel Speicherplatz und Prozessorzyklen für die Recovery-Line-Berechnung verbraucht werden, kann koordiniertes Checkpointing in einer hohen Anzahl an Netzwerknachrichten resultieren, um die jeweiligen Teilsequenzen, siehe Kapitel 3.4.4.2, anzusteuern.

Im umgekehrten Fall kann das System die Ausführung des Checkpointingprotokolls behindern. Ist der *swap*-Bereich ausgereizt, kann die Reaktionsfähigkeit des Systems beeinträchtigt werden, was dazu führt, dass auch das Protokoll nicht effizient abgearbeitet werden kann. Steht kein ausreichend großer Festplattenspeicher zur Abbildsicherung zur Verfügung, wird das Protokoll nicht erfolgreich beendet. Stehen zu wenig Prozessorzyklen bereit, kann, in Abhängigkeit des Protokolls, die Anwendungsausführung längerfristig beeinträchtigt werden.

Wird also eine Checkpointingaktion nicht mit dem Anwendungsverhalten abgestimmt, kann es dazu kommen, dass System und Anwendung negativ beeinflusst werden. Wird beispielsweise inkrementell gesichert, obwohl der Anwendungsadressraum zwischen zwei Checkpoints größtenteils modifiziert wurde, entsteht ein Mehraufwand gegenüber einer vollständigen Sicherung, siehe Kapitel 7.5. Um sich für inkrementelles Sichern zu entscheiden, muss das Anwendungsschreibzugriffsmuster zuvor überwacht worden sein.

Aus diesen Beispielen wird ersichtlich, dass *eine ideale Checkpointingstrategie aus Informationen des System- und Anwendungsverhaltens abgeleitet werden muss.* Verhaltensinformationen können anhand von Überwachungsmonitoren wie Ganglia [98] ermittelt werden, welche Daten individueller Ausführungskontexte erheben. Die fortwährende Überwachung verursacht jedoch einen Zusatzaufwand und kann damit zur Performanzbeeinträchtigung führen.

Alternativ zur System- und Anwendungsüberwachung werden unter [112] Ressourcen-Fehlerindizes eingebunden, um relevante Checkpointingparameter daran auszurichten. In [113] [74] werden mithilfe mathematischer Ansätze Mean Time To Failure (MTTF)-Werte auf Basis von Wahrscheinlichkeits-Werten bestimmt.

7.3 Dimensionen adaptiven Checkpointings

Die am weitesten verbreitete Technik, um das Checkpointingverhalten anzupassen, besteht darin, Checkpointing-Intervalle zu modifizieren, abhängig von der aufgezeichneten Fehlerrate. Die Gefahr, dass wertvolle Zwischenzustände verloren gehen, wird vermindert, indem die *Checkpointingfrequenz erhöht wird*. Der Checkpointaufwand wird reduziert, indem beispielweise die *Checkpointingfrequenz reduziert wird*.

Findet Checkpointing und Restart nicht auf demselben Knoten statt, muss gewährleistet werden, dass die benötigten Abbilder beim Restart verfügbar sind. Letzteres wird erreicht, indem Abbilder gezielt auf bestimmten Checkpoint-Servern platziert, oder in einem verteilten Dateisystem repliziert werden. Beides verursacht einen Aufwand und kann daher als Checkpointingparameter gesteuert werden.

Ein-/Ausgabe-Zeiten verschiedener Speichermedien variieren und tragen entscheidend zum Aufwand einer Checkpointing/Restart-Aktion bei, siehe Kapitel 8. Abbilder können jedoch auch im Hauptspeicher, anstelle auf Festplatte, abgelegt werden, um von den kürzeren Speicherzugriffszeiten zu profitieren. In [106] wird das Koheränz-Protokoll einer Cache-Only-Memory-Architektur abgeändert, um Restart-Daten in replizierter Form im Speicher zu halten.

In Abhängigkeit des verfügbaren Speichers können Abbilder komprimiert werden. Dies erfordert jedoch zusätzlichen Berechnungsaufwand, sodass sich die Checkpoint-Aktion zeitlich verlängert.

Logging-basierte Recovery-Verfahren werden darin unterschieden, zu welchem Zeitpunkt Determinanten auf stabilem Speicher aufgezeichnet werden, siehe Kapitel 1.2.3. In Abhängigkeit der Speichergeschwindigkeit kann pessimistisches Logging verwendet werden, um Determinanten synchron aufzuzeichnen, bei optimistischem Logging wird asynchron gespeichert. Vorher muss evaluiert werden, welche Auswirkungen Orphan-Prozessse, die bei optimistischem Logging eintreten können, auf die Anwendungsausführung haben können.

Dass sich koordiniertes und unkoordiniertes Checkpointing mit pessimistischem Logging unterschiedlich verhalten, wurde in [23] aufgezeigt. Im folgenden Kapitel 7.4 wird jedoch betrachtet, was beachtet werden muss, wenn zur Laufzeit, ohne die Anwendung anzuhalten, zwischen koordiniertem und unkoordiniertem Checkpointing gewechselt wird. Dieser Ansatz unterscheidet sich von der in der Literatur vorherrschenden Methodik, lediglich Parameter *eines* Checkpointingprotokolls zu modifizieren.

7.4 Wechsel des Checkpointingprotokolls

Ein Checkpointingprotokollwechsel muss für alle Job-Einheiten eines Jobs zu einem Zeitpunkt vorgenommen werden. Mit anderen Worten, die parallele, beziehungsweise überlappende Ausführung verschiedener Checkpointingprotokolle durch mehrere Job-Einheiten

muss vermieden, weil Checkpointingprotokolle inkompatibel zueinander sind. Beispielsweise entsteht eine Behinderung, wenn unabhängig gesichert, gleichzeitig Prozesse aber auch synchronisiert werden. Andererseits müssen keine Metadaten an Nachrichten angehangen und mit den Nachrichten bei koordiniertem Checkpointing aufgezeichnet werden. Weiterhin führt *die Kombination von Abbilddateien verschiedener Checkpointingprotokolle*, wie jene

- mit und ohne abgespeicherten Determinanten,
- mit und ohne aufgezeichneten Nachrichten,

zu Inkonsistenz, wenn verschiedene Wiederherstellungsmethoden zu einem Zeitpunkt miteinander kombiniert ausgeführt werden.

Idealerweise wird zwischen verschiedenen Checkpointingprotokollen gleichzeitig auf allen involvierten Rechnern umgeschaltet[3]. Aufgrund der fehlenden Uhrensynchronisation kann letzteres nicht gewährt werden. Dadurch kann es *knotenübergreifend an den Protokollübergängen* zu inkompatiblen Abbildern und inkompatiblen Wiederherstellungsvorgängen kommen.

Im Folgenden wird beschrieben, was passiert und korrigiert werden muss, wenn zwischen koordiniertem und unkoordiniertem Checkpointing umgeschalten wird, um *gezielt*, in Abhängigkeit des dynamischen Ausführungskontextes, von den Vorteilen koordinierten und unkoordinierten Checkpointings zu profitieren. Der Wechsel zu unkoordiniertem Checkpointing ist sinnvoll, wenn

- der damit verbundene Aufwand während der fehlerfreien Ausführung kleiner als der Synchronisierungsaufwand koordinierten Checkpointings ist,
- maximale Autonomie über das Checkpointing gefordert wird,
- die Gefahr des Domino-Effekts ausgeschlossen werden kann[4].

Konkret wird unkoordiniertes Checkpointing attraktiv, wenn viele Prozesse existieren, weil keine aufwendige Prozess-Synchronisierung vorgenommen werden muss. Schneller Hauptspeicher begünstigt zudem, dass Abhängigkeitsinformationen im fehlerfreien Betrieb schnell abgespeichert werden können.

Der Wechsel zu koordiniertem Checkpointing ist sinnvoll, wenn

- der Aufwand unkoordinierten Checkpointings im fehlerfreien Betrieb sehr groß ist, beispielsweise aufgrund des langsamen Speichers,
- Unterbrechungen der Anwendung, infolge der Prozess-Synchronisierung, toleriert werden können,

[3] In Bezug auf die GCA ist der Job-Checkpointer verantwortlich dafür, einen Strategiewechsel zu initiieren, weil er alle Job-Einheiten eines Jobs kennt.

[4] Das wird realisiert, indem die Recovery-Line periodisch berechnet wird und daraus Rückschlüsse über relevante Checkpointingaktionen gezogen werden.

- die Wahrscheinlichkeit des Dominoeffekts sehr hoch ist.

Analog zu oben ist koordiniertes Checkpointing attraktiv, wenn viele abhängigkeitserzeugenden Nachrichtenkommunikationen auftreten. Letzteres erhöht den Aufwand der Recovery-Line-Berechnung und kann dazu führen, viele Job-Einheiten zurück zu rollen.

7.4.1 Der Wechsel von unkoordiniertem zu koordiniertem Checkpointing

Findet der Übergang zuerst bei einem Sender- oder zuerst bei einem Empfängerprozess statt, entstehen zwei charakteristische Fälle, weil Abhängigkeitsinformationen bei unkoordiniertem und koordiniertem Checkpointing unterschiedlich verarbeitet werden.

Im Folgenden wird davon ausgegangen, dass Abhängigkeitsinformationen an Nachrichten angehangen und nicht im verteilten Dateisystem abgelegt werden, bevor diese versendet werden.

7.4.1.1 Der Senderprozess wechselt vor dem Empfängerprozess

In Abbildung 7.1 wechselt P1 vor P2 von unkoordiniertem zu koordiniertem Checkpointing und sendet Prozess P2 zwei Nachrichten.

Da P1 im Kontext koordinierten Checkpointings Anwendungsnachrichten keine Abhängigkeitsinformationen versendet, können letztere von P2, bei Empfang von Nachricht m1, nicht aufgezeichnet werden. Daher wird das Empfangsereignis von m1 *nicht* in die Recovery-Line-Berechnung einbezogen. Wird nicht $S1_2$, jedoch $S2_2$ verwendet, entsteht eine Waisen-Nachricht.

Eine weitere Eigenschaft dieses Szenarios muss berücksichtigt werden. Da nicht jeder Prozess das Signal zum Protokoll-Wechsel zur gleichen Zeit erhält, jedoch unmittelbar ein koordinierter Checkpoint initiiert werden kann, können in einem Kanal Nachrichten mit und ohne Abhängigkeitsinformationen enthalten sein. Bei Kanal-Leerung, im Kontext koordinierten Checkpointings, müssen jedoch beide Nachrichtentypen, insbesondere jene mit Abhängigkeitsinformationen, aus dem Kanal entfernt, gesichert und bei Restart wieder eingespielt werden können.

7.4.1.2 Der Empfängerprozess wechselt vor dem Senderprozess

In Abbildung 7.2 empfängt P2 die Nachricht m1 und ist im Modus koordinierten Checkpointings. Hingegen sendet P1 m1 im Modus unkoordinierten Checkpointings.

Wird m1 empfangen, können die enthaltenen Abhängigkeitsinformationen ignoriert werden, da der bevorstehende koordinierte Checkpoint ohnehin eine Synchronisierung aller

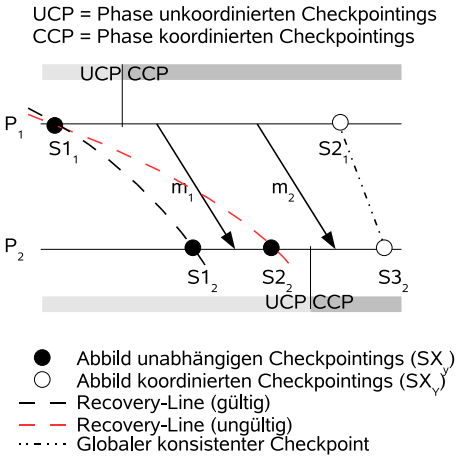

Abbildung 7.1: Fehlende Abhängigkeitsinformationen bei m1

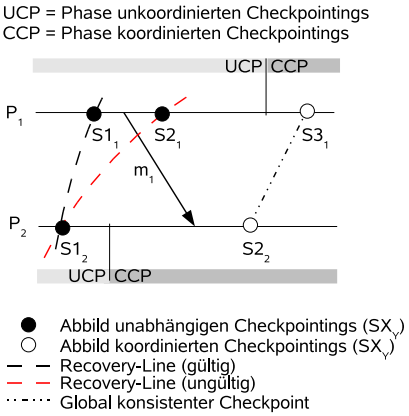

Abbildung 7.2: Ignorierbare Abhängigkeitsinformationen

involvierten Prozesse bedarf, deren Abhängigkeiten dabei inhärent aufgelöst werden, beispielsweise indem in-Transit Nachrichten herausgeschoben werden. Bei einem Restart von einem Abbild koordinierten Checkpointings werden keine zuvor aufgezeichneten Abhängigkeitsinformationen benötigt. Jedoch entsteht im fehlerfreien Betrieb zusätzlicher Aufwand, wenn mitgelieferte Zusatzinformationen herausgefiltert werden müssen. Erfolgt dies nicht, können Inkonsistenzen entstehen, wenn die ungefilterten Daten an die Anwendung ausgeliefert werden.

Die Recovery-Line, welche $S2_1$ mit $S1_2$ verbindet, ist ungültig, da hierdurch eine verlorene Nachricht beim Restart entsteht.

7.4.2 Der Wechsel von koordiniertem zu unkoordiniertem Checkpointing

7.4.2.1 Der Sender wird vor dem Empfänger von einem Wechsel benachrichtigt

In Abbildung 7.3 ist der Sender von Nachricht m1 bereits im Modus unabhängigen Checkpointings, der Empfänger im Modus koordinierten Checkpointings. Wird m1 empfangen, werden daher keine Abhängigkeitsinformationen aufgezeichnet. Sie müssen jedoch aus m1 herausgefiltert werden, bevor die Daten der Anwendung übergeben werden können.

Verwendet P1 $S2_1$ und P2 $S2_2$ im Kontext eines Restarts, entsteht eine Orphan-Nachricht. $S2_1$ und $S1_2$ können nicht verwendet werden, da hierbei Abbilder verschiedener Checkpointprotokolle vorliegen, die individueller Wiederherstellungsabläufe bedürfen, die inkompatibel zueinander sind.

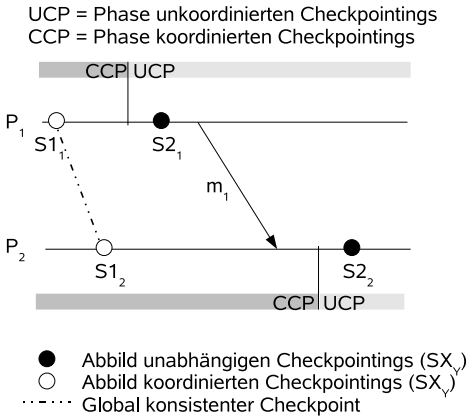

Abbildung 7.3: Abhängigkeitsdaten werden ignoriert

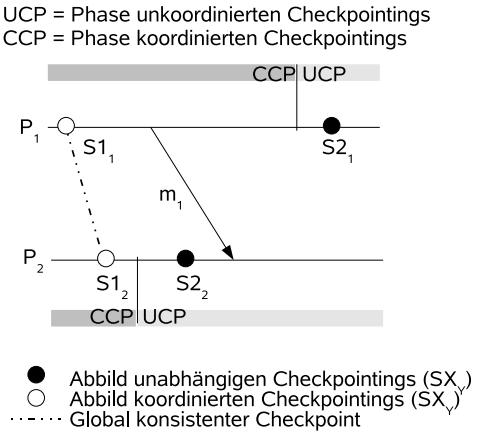

Abbildung 7.4: Abhängigkeitsdaten werden nicht gesendet

7.4.2.2 Der Empfänger wird vor dem Sender von einem Wechsel benachrichtigt

In Abbildung 7.4 ist der Empfänger bereits im Modus unabhängigen, der Sender noch im Modus koordinierten Checkpointings. Wird die Nachricht m1 empfangen, können keine Abhängigkeitsinformationen herausgefiltert und gespeichert werden. Dies führt bei der Recovery-Line-Berechnung, auf Basis von $S2_1$ und $S2_2$ dazu, dass die m1-Abhängigkeit ausgeblendet wird, sodass beim Restart eine verlorene Nachricht entsteht[5]. Die Zustände $S2_1$ und $S2_2$ werden fälschlicherweise als global konsistenter Zustand ausgewiesen, der aufgrund

[5]Obwohl keine Abhängigkeitsinformatioenn mitgeschickt werden, kann zumindest der Empfang der Nachricht vermerkt werden, um die Existenz einer Abhängigkeit auszuweisen.

der verlorenenen Nachricht inkonsistent ist.

7.4.3 Konflikte bei Protokollübergängen im Überblick

Folgende Konflikte entstehen:

1. Fall: Werden keine Abhängigkeitsinformationen gesendet, entsteht ein fehlerhafter Ausgangspunkt für die Recovery-Line-Berechnung, sodass bei einem Restart Waisen-Nachrichten auftreten können, siehe Kapitel 7.4.1.1.
2. Fall: Werden gesendete Abhängigkeitsinformationen nicht herausgefiltert, kann eine Inkonsistenz bei der Nachrichtenauslieferung an die Anwendung auftreten, siehe Kapitel 7.4.1.2.
3. Fall: Werden gesendete Abhängigkeitsinformationen nicht aufgezeichnet, kann es beim Restart zu Waisen-Nachrichten kommen, siehe Kapitel 7.4.2.1.
4. Fall: Werden keine Abhängigkeitsinformationen gesendet, kann es beim Restart zu verlorenen Nachrichten kommen, siehe Kapitel 7.4.2.2.
5. Fall: Existiert kein Bezug zwischen Abbildern verschiedener Protokolle, können Zwischenzustände verloren gehen.

7.4.4 Lösung

7.4.4.1 Netzwerkfunktionen, die mehrere Checkpointingprotokolle unterstützen

Fälle 1,2, 3 und 4 lassen sich auf dieselbe Ursache zurückführen: bestehende Netzwerkfunktionen der nativen Standard C-Bibliothek bieten bisher keine Unterstützung für mehrere Checkpointingprotokolle.

Fall 1 und 4 ähneln einander, weil hierbei zusätzlich Abhängigkeitsinformationen gesendet werden müssen. Letzteres kann erzielt werden, indem die native *send*-Funktion modifiziert wird. Weil in Fall 1 der Sender bereits im Modus koordinierten Checkpointings ist, muss er solange Abhängigkeitsinformationen senden, bis auch der Empfänger im Modus koordinierten Checkpointings ist. Weil in Fall 4 der Empfänger bereits im Modus unkoordinierten Checkpointings ist, muss der Sender Abhängigkeitsinformationen mitschicken.

Fall 2 und 3 ähneln einander, weil hierbei die mitgeschickten Abhängigkeitsinformationen von den Nutzdaten getrennt werden müssen. Letzteres kann erreicht werden, indem die native *recv*-Funktion modifiziert wird. Bei Fall 2 müssen Abhängigkeitsinformationen auf Empfängerseite solange von Nutzdaten getrennt werden, bis der Sender auch im Modus koordinierten Checkpointings ist. Bei Fall 3 müssen Abhängigkeitsinformationen auf

Empfängerseite solange von Nutzdaten getrennt werden, bis der Empfänger selbst im Modus unkoordinierten Checkpointings ist. Darüber hinaus müssen diese Daten aufgezeichnet werden, um Waisen-Nachrichten zu vermeiden.

Die send und recv-Funktionen verhalten sich beim Protokollübergang relativ zum aktuellen Checkpointingprotokollmodus des Senders und Empfängers. Das Verhalten der *recv*-Funktion eines Empfängerprozesses kann entsprechend angepasst werden, indem der Checkpointingprotokollmodus des Empfängerprozesses mit dem des Senderprozesses gleichgesetzt wird. Der Sender trägt hierzu den eigenen Checkpointingprotokollmodus im Nachrichten-Header ein, welcher vom Empfänger ausgelesen wird und fortan das *recv*-Verhalten bestimmt.

Eine Anpassung der *send*-Funktion ist komplizierter, weil eine Rückkommunikation, vom Empfänger zum Sender, notwendig ist. Hierzu ist ein separater Signalisierungsmechanismus notwendig, welcher mithilfe zusätzlicher Steuerkanäle integiert werden muss. Letzteres muss transparent für einen Anwendungsprogrammierer geschehen, da nicht vorausgesetzt werden kann, dass dieser über Fehlertoleranzwissen verfügt. Um Signalisierungsverzögerungen zu reduzieren, müssen außerhalb der Reihenfolge empfangbare (engl. out-of-band) Signalnachrichten eingesetzt werden. Nach dem Wechsel zum koordinierten Checkpointing, fährt der Sender in Fall 1 damit fort, Abhängigkeitsinformationen zu senden, bis er das Signal des Empfängers, über dessen Wechsel zum koordinierten Checkpointing, erhalten hat. In Fall 4 muss der Sender vom Empfänger über ein Signal dazu aufgefordert werden, jede weitere zu versendende Nachricht mit Abhängigkeitsinformationen zu versehen.

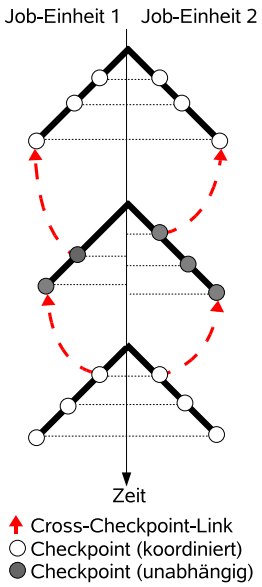

Abbildung 7.5: Protokollübergreifende Checkpointverwaltung

7.4.4.2 Protokollübergreifende Checkpointverwaltung

Bei adaptivem Checkpointing können die Phasen verschiedener Checkpointing-Protokolle sequentiell angeordnet sein. Falls eine Phase des koordinierten einer Phase des unabhängigen Checkpointings vorausgeht, darf bei einer erfolglosen Recovery-Line-Berechnung kein Rückfall auf den initialen Zustand erfolgen. Hingegen muss auf den letzten koordinierten Checkpoint der vorangehenden Phase referenziert werden, da gesicherte Zwischenzustände dem initialen Zustand vorzuziehen sind. Um in der Vergangenheit liegende Zustände gezielt herzustellen, müssen auch Abbilder unkoordinierten von denen koordinierten Checkpointings aus referenziert werden können.

Da sich kein Checkpointingprotokoll über eine Kooperation mit einem anderen definiert, müssen Checkpointing-Dateien verschiedener Protokolle (siehe Kapitel 7.4) protokollübergreifend miteinander in Bezug gesetzt werden. Dies ist die Aufgabe des Job Checkpointers, da er auf höchster Ebene über Protokollwechsel entscheidet und alle Job-Einheiten kennt.

Abbildung 7.5 stellt die Grundidee der Abbildverwaltung für eine aus zwei Job-Einheiten bestehenden Anwendung dar. Jedes Zeitintervall, in dem ein Checkpointprotokoll pro Job-Einheit verwendet wird, wird als *Zweig* bezeichnet. Ein Zweig kann ein oder mehrere Abbilder des gleichen Protokolltyps enthalten. Bei koordiniertem Checkpointing werden Abbilder *aller* Job Einheiten, bei unabhängigem Checkpointing wird das Abbild *einer einzelnen* Job Einheit zu einem logischen Zeitpunkt erstellt. Das letzte Abbild eines Zweiges wird als Blatt bezeichnet. Über rückwärtsgerichtete Verweise kann von einem Zweig aus das Blatt des Vorgängerzweigs eines Anwendungsteils referenziert werden.

Eine Garbage Collection muss Informationen dieser Abbildverwaltung einbeziehen und diese gegebenenfalls modifizieren. Werden Abbilder gelöscht, muss die protokollübergreifende Verkettung eingehalten und angepasst werden.

7.5 Inkrementelles Checkpointing

7.5.1 Überblick

Mit einer reduzierten Abbildgröße ergibt sich meist ein Geschwindigkeitsvorteil beim Checkpointing. Dies kann erreicht werden, indem ausschließlich seit dem letzten Sicherungspunkt veränderte Speicherinhalte in den neuen Sicherungspunkt aufgenommen werden.

Die zentrale Herausforderung inkrementellen Checkpointings besteht darin, *veränderte Speicherinhalte zu erkennen*. Beim Speicherseiten-basierten Ansatz können veränderte Inhalte zum Checkpoint-Zeitpunkt schnell identifiziert werden, mithilfe der zugrundeliegenden Memory Management Unit (MMU).

In modernen Betriebssystemen wird der verfügbare Hauptspeicher in Einheiten fester Größe, sogenannten Kacheln, eingeteilt. *Paging* bezeichnet die Abbildung logischer Seiten eines Prozesses auf Kacheln, sodass ein Prozess virtuell mehr Speicher adressieren kann, als physikalisch, zu einem gegebenen Zeitpunkt, vorhanden ist und vor unbefugtem Überschreiben des eigenen Adressraums durch andere Prozesse geschützt wird. Im Kern wird diese Abbildung pro Prozess anhand einer zweistufigen (x86), dreistufigen (ia64), beziehungsweise vierstufigen (x86_64) Tabellenstruktur, bestehend aus Seitenverzeichnis, Seitentabelle und Hardware-abhängigen, zusätzlichen Ebenen, verwaltet. Pro Seitentabelleneintrag existieren zwei Bit-Flags, welche die Zustandsinformationen einer Seite widerspiegeln, die im Kontext inkrementellen Checkpointings wichtig sind.

Wird inkrementelles Checkpointing aufbauend auf Paging realisiert, dürfen

1. existierende, systemkritische Dienste, wie die Speicherverwaltung des Betriebssystems, nicht beeinträchtigt werden,
2. für zukünftige Zwecke reservierte Bits in den Seitentabelleneinträgen nicht für Checkpointing verwendet werden,
3. keine, beziehungsweise nur geringe Leistungseinbußen entstehen,
4. keine extensiven Änderungen am Betriebssystemquelltext vorgenommen werden.

Bei der Implementierung inkrementellen Checkpointings tritt deutlich zu Tage, dass einige dieser Aspekte im Widerspruch zueinander stehen. Beispielsweise können modifizierte Inhalte mithilfe Pagings anwendungstransparent und performant erkannt werden. Andererseits bedingt dies, dass Paging-relevante Strukturen modifiziert werden müssen, sodass bestehende Betriebssystemdienste beeinträchtigt werden. Hingegen müssen zentrale Kernstrukturen abgeändert werden, damit nur minimale Performanzeinbußen entstehen, wenn modifizierte Seiten erkannt werden sollen.

Es ist dennoch wichtig, inkrementelles Checkpointing auf Betriebssystemebene zu realisieren, da hierdurch die unter 2.2.5 erwähnte GCA-Anforderung der Anwendungstransparenz erfüllt wird.

7.5.2 Modifizierte Speicherseiten

Im Folgenden werden der Dirty-Bit- und der Write-Bit-basierte Ansatz, zur Identifizierung veränderter Seiten, erläutert.

7.5.2.1 Das Dirty-Bit

Schreibender Zugriff auf eine Seite bewirkt, dass die MMU das Dirty-Bit im zugehörigen Seitentabelleneintrag setzt. Das Dirty-Bit wird jedoch nicht von der Paging Unit der MMU zurückgesetzt. Dies obliegt der Verantwortung des Betriebssystems.

Wird das Dirty-Bit für Checkpointing verwendet, beeinflusst es unmittelbar die Systemstabilität (Fall 1). In umgekehrter Richtung gilt, wird das Dirty-Bit bereits von der Systemsoftware genutzt, kann es inkrementelles Checkpointing beeinflussen, sodass inkonsistente Checkpoints entstehen (Fall 2).

Fall 1: Veränderte Speicherinhalte müssen auch *nach* einem Checkpoint *erneut* erkannt werden können.

Letzteres erfordert, dass das Dirty-Bit *explizit nach einem Checkpoint zurückgesetzt wird.* Das ist jedoch problematisch, da das Dirty Bit-Flag von Betriebssystemkomponenten, die *keinen Bezug zu Checkpointing besitzen,* verwendet wird, beispielsweise dem Page Frame Reclaiming Algorithm (PFRA) in Linux, siehe [25]. Das System benötigt eine minimale Menge freier Kacheln, um lauffähig zu bleiben. Wird daher ein gewisser Schwellenwert freier Kacheln unterschritten, beginnt der PFRA damit, Seiten zu markieren (Dirty-Bit setzen)[6] und veranlasst, diese, in verzögerter Form, auszulagern. Weiterhin werden zu synchronisierende Seiten, wie Kernel Cache Seiten (Page Cache) vom PFRA individuell behandelt, [25]. Wird im Kontext inkrementellen Checkpointings das Dirty-Bit zurückgesetzt, wird damit die Grundlage zerstört, um über notwendige Auslagerungen und Synchronisierungen zu entscheiden. Das in Kapitel 7.5.1 beschriebene Kriterium 1 wird hierdurch verletzt. [7]

Fall 2: Manipulationen des Dirty-Bits, seitens des Betriebssystems, bestimmen darüber, ob eine Seite gesichert wird.

Modifizierte Seiten des Page Caches werden mit Dateisystemblöcken auf einer Festplatte in gewissen Abständen synchronisiert. Das impliziert, dass das Dirty-Bit zurückgesetzt werden muss. Das sich hier ergebende Problem besteht darin, dass *Synchronisierungen* von Kacheln mit der Festplatte *unabhängig von inkrementellem Checkpointing vonstatten gehen.* Damit kann ein gesetztes Dirty-Bit (nach Schreibzugriff auf eine Seitenadresse) *vor* dem nächsten inkrementellen Checkpoint vom PFRA zurückgesetzt werden. Dies führt zum Checkpointzeitpunkt dazu, dass eine modifizierte Seite nicht erkannt wird und ein inkonsistenter Checkpoint entsteht.

[6] Bei schreibendem Seitenzugriff und Seitenauslagerung wird das Dirt-Bit gesetzt.

[7] Falls Swapping ausgeschalten und Datei-Mapping ausgeschlossen wird, kann das Dirty Bit-Flag zu Checkpointing-Zwecken verwendet werden, ohne System-kritische Dienste zu beeinflussen.

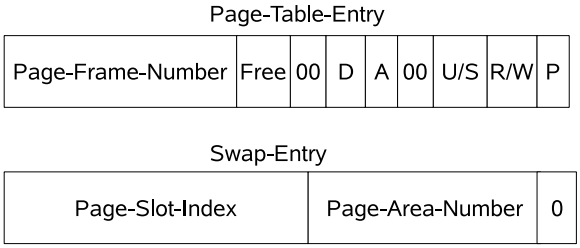

Abbildung 7.6: Inhalte eines Seitentabelleneintrages

7.5.2.2 Das Write-Bit

Für jede Seite existieren zwei Zugriffsrechte (Lesen und Schreiben), die im Write-Bit in jedem Seitentabelleneintrag gespeichert werden. Ist der Bitwert gleich 0, kann die entsprechende Seite gelesen, bei Wert 1 geschrieben und gelesen werden. Wird schreibend auf eine nur mit Leserechten versehene Seite zugegriffen, entsteht ein Schreibschutzseitenfehler (engl. protection fault). Dies ist eine Ausnahme (engl. exception), welche von der MMU erkannt wird. In Folge löst die Ausnahmebehandlungsroutine den Schreibschutz auf, das Write-Bit wird gesetzt. Das Write-Bit kann zusätzlich noch mithilfe des *mprotect* Aufrufs vom Benutzeradressraum aus sowie im Kern gesetzt werden.

Das Write-Bit wird beim inkrementellen Checkpointing verwendet, um modifizierte Seiten zu erkennen, indem es pro Seite nach jedem Checkpoint explizit zurückgesetzt wird. Findet ein Seitenschreibzugriff im Intervall zwischen zwei inkrementellen Checkpoints statt, wird das Write-Bit gesetzt. Falls es beim nächsten Checkpoint noch gesetzt ist, indiziert es eine zwischenzeitliche Seitenänderung, sodass die assoziierte Seite in den Checkpoint aufgenommen werden muss.

Der Vorteil dieses Verfahrens gegenüber dem Dirty-Bit-Ansatz besteht darin, dass veränderte Seiten größtenteils, *mprotect* benutzende Anwendungen ausgenommen (siehe Kapitelende), erkannt werden, ohne dass vom PFRA-initiierte systemkritische Auslagerungen verhindert werden und damit Systemstabilität gefährdet ist. Jedoch entsteht ein Konflikt mit dem PFRA hinsichtlich Konsistenz. Wird eine Seite beschrieben, anschließend in eine sogenannte Swap Area auf der Festplatte ausgelagert, wird der zugehörige Seitentabelleneintrag durch einen sogenannten Swap-Entry ausgetauscht, siehe Abbildung 7.6. Beim Swapping erhalten die Bits des Seitentabelleneintrags eine Swapping-spezifische Bedeutung. Das erste bis 31. Bit wird als Festplattenadresse der ausgelagerte Seite verwendet, um sie bei der Wiedereinlagerung korrekt adressieren und somit einlesen zu können (7 Bit Swap Area Nummer, 24 Bit Page Slot Index einer Swap-Area). Bei Wiedereinlagerung, infolge eines Seitenzugriffs, wird der Seitentabelleneintrag aktualisiert. Nach einem schreibenden Zugriff wird das Dirty und Write-Bit gesetzt. Ein Lesezugriff hingegen resultiert darin, dass ein vor dem Auslagern gesetztes Write-Bit zurückgesetzt wird. Damit wird eine geänderte Seite nicht mehr als solche erkannt. Der folgende Checkpoint integriert diese Seite nicht und wird damit inkonsistent. Dieser Spezialfall erfordert, dass das Be-

triebssystem abgeändert wird, um inkrementelles Checkpointing bei Swapping vollständig zu unterstützen. Beispielsweise kann im Modus inkrementellen Checkpoitnings der erste Seitenleseaufruf nach einer Seitenwiedereinlagerung in einen Schreibaufruf transformiert werden.

Umgekehrt beeinflusst inkrementelles Checkpointing den *PFRA* nicht, wenn dabei Swapping berücksichtigt wird. Zwei Fälle sind dabei abzudecken. Einerseits kann eine Seite vor ihrer Auslagerung noch verändert worden sein. Dann muss sie explizit eingelesen und mitgesichert werden. Andererseits, wurde sie nicht verändert, darf das Write-Bit nach einem Checkpoint nicht ohne Weiteres zurückgesetzt werden. Wird das Bit einer ausgelagerten Seite, welches außerhalb des Swap-Kontextes dem Write-Bit entspricht, verändert, wird die, für die Wiedereinlagerung wichtige Auslagerungsadresse, manipuliert. Ausgelagerte Seiten können jedoch anhand des Seitentabelleneintrags erkannt werden, wenn das Present Bit nicht gesetzt ist und mindestens ein Bit der verbleibenden 31 Bits gesetzt ist. Lezteres ist darauf zurückzuführen, dass keine Seite im ersten Page-Slot einer Swap-Area liegen darf, jedoch Verwaltungsinformationen.

An dieser Stelle wird ersichtlich, dass herausgefunden werden muss, *ob die Seite, über Swappings hinweg, verändert wurde oder nicht*. Dieser Aufwand lohnt, weil explizites Einlesen von Seiten zu aufwendig ist.

Copy-On-Write (COW) stellt einen weiteren Software-Mechanismus dar, der in Verbindung mit dem Write-Bit steht und betrachtet werden muss. Hierbei wird der Elternprozess-Adressraum, nach einer Kindprozess-Erzeugung, nicht vollständig und nicht unmittelbar kopiert, sondern erst dann, wenn entweder Eltern- oder Kindprozess schreibend auf eine gemeinsame Seitenadresse zugreift. Bis dahin arbeiten beide lesend auf gemeinsamen Kacheln.

Bei inkrementellem Checkpointing muss ermittelt werden, ob *COW* aktiv wird, wenn das Write-Bit gesetzt wird, insbesondere wenn es wiederholt, über mehrere Checkpoints hinweg, gesetzt wird. Trifft letzteres zu, findet eine ungewollte Seitenreplizierung statt, welche bei Seitengrößen von 4 MB zu hohem Aufwand führt. Wird eine gemeinsame Prozessseite vom Vater oder Kind, nach Kindprozess-Erzeugung, erstmalig beschrieben, wird ein Seitenfehler ausgelöst. Zuvor gilt der zugehörige Seitentabelleneintrag des Kind-Prozesses als nicht initialisiert. In Folge füührt die *__do_fault*[8] Handlerfunktion dazu, dass die Seite kopiert wird. Anschließend wird ein spezielles Bit (anonymous Bit im mapping-Feld des Seitendeskriptors) gesetzt und der Seitentabelleneintrag im Kindprozess initialisiert. Wird eine Version der Seite hingegen erneut schreibgeschützt (Write-Bit wird nach Checkpoint zurückgesetzt) und beschrieben, wird sie nicht mithilfe von *__do_fault* kopiert, sondern die Funktion *do_wp_page* aufgerufen, welche für die Seite überprüft, ob das spezielle Bit gesetzt ist. Ist es bereits gesetzt, wird sie nicht erneut kopiert.

Weil ein wiederholt gesetztes Write-Bit nicht zu einer ungewollten Seitenreplizierung führt, existieren keine Interferenzen zwischen *COW* und inkrementellem Checkpointing. Dieser

[8] Siehe http://lxr.linux.no/#linux+v2.6.32/mm/memory.c#L2699

Sachverhalt trifft auch für Szenarien zu, in denen Seiten von mehreren Prozessen gemeinsam verwendet werden (MAP_SHARED), da *COW* hierbei nicht genutzt wird. Das Kriterium 1 wird in Verbindung mit *COW* nicht verletzt.

Wird das Write-Bit verwendet, um modifizierte Seiten zu erkennen, entsteht unter anderem ein Performanzverlust, da hierdurch zusätzliche Seitenfehler provoziert werden, wenn das Write-Bit zurückgesetzt wird. Weil die Performanz beeinträchtigt wird, tritt eine Verletzung des 3. Kriteriums ein.

Eine zusätzliche Herausforderung entsteht im Zusammenhang mit dem *mprotect* Aufruf. Hierdurch können Zugriffrechte einer Speicherseite gesetzt werden, indem von der Benutzerebene aus das Write-Bit modifiziert wird. Mithilfe von mprotect kann eine als verändert markierte Seite nicht als verändert indiziert werden, wenn das Write-Bit im Kern überprüft wird. Diese Interferenz führt ebenfalls zu inkonsistenten Abbildern.

Um den Write-Bit-Ansatz zu bewerten, muss die Gewichtung eingehaltener und verletzter Kriterien berücksichtigt werden. Obwohl performantes inkrementelles Checkpointing ideal und wünschenswert ist, besitzt die Umsetzbarkeit inkrementellen Checkpointings eine existentiellere Bedeutung.[9] Das 1. Kriterium (veränderte Inhalte müssen ohne Einschränkungen bestehender Software-Mechanismen erkannt werden) wird im Hinblick auf *mprotect* verletzt. Hierbei muss jedoch erwähnt werden, dass nur ein Bruchteil aller Anwendungen von diesem Aufruf Gebrauch macht. In der, um Checkpointing erweiterten, JSDL-Datei einer Anwendung, siehe Kapitel 3.2.1.7, kann vermerkt werden, ob inkrementelles Checkpointing angewendet werden darf, beziehungsweise ob *mprotect* verwendet wird. Wird Swapping, wie oben skizziert, berücksichtigt, gibt es keine Interferenzen mit dem Write-Bit-Ansatz. Andere systemkritische Dienste werden nicht beeinflusst.

Das 2. Kriterium (für zukünftige Zwecke reservierte Elemente dürfen nicht für Checkpointing verwendet werden) wird eingehalten, da kein reserviertes Bit verwendet wird, dessen Nutzung in Zukunft zum jetzigen Zeitpunkt unklar ist. Das Write-Bit hingegen wird bereits seit der Einführung des Pagings verwendet. Die mit Swapping in Zusammenhang stehenden Speziallfälle müssen jedoch durch Abänderung des Betriebssystem behandelt werden.

Das 3. Kriterium (es darf nur niedriger Aufwand für das Betriebssystem entstehen, veränderte Seiten zu erkennen) wird eingehalten. Bei Swapping kann Mehraufwand entstehen, wenn explizit eingelagert wird. Swapping kann man jedoch mit einer Hauptspeicheraufrüstung vermeiden.

Das 4. Kriterium (es sollen keine umfangreichen Änderungen am Betriebssystem-Quelltext vorgenommen werden, um inkrementelles Checkpointing zu ermöglichen) wird im Bezug auf die Write-Bit-Rücksetzung und Swappingunterstützung verletzt.

[9] Das gilt nur, insofern die Gesamtkosten einer inkrementellen Sicherung nicht die Kosten einer vollständigen Sicherung übersteigen. Letzteres kann eintreten, da aufgrund der Verwaltung zusätzlicher Metadaten ein inkrementeller Checkpoint länger als ein vollständiger Checkpoint dauern kann.

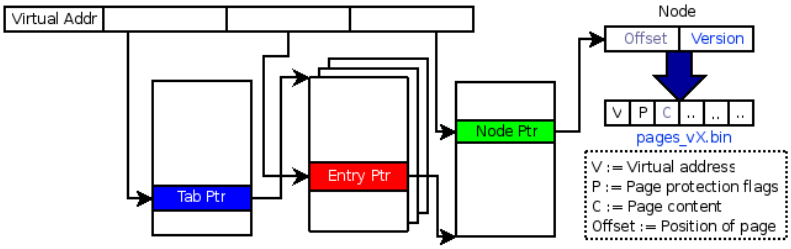

Abbildung 7.7: Buchführungsstruktur veränderter Seiten

Im Folgenden wird dargestellt, wie inkrementellen Checkpointings in das Linux-Betriebssystem integriert werden. Hierbei wird der Write-Bit Ansatz zugrunde gelegt, da die Nachteile des Dirty-Bit-Ansatzes schwerwiegender sind, jedoch eine kernbasierte Lösung aus Performanzgründen angestrebt wird.

7.5.3 Verwaltung modifizierter Speicherseiten

Die Seiten einer Anwendung können über mehrere inkrementelle Checkpoint-Abbilder verteilt sein. Zum Zeitpunkt eines Neustarts muss die aktuellste Version jeder Anwendungsseite *schnell* aus einer Menge von Abbildern identifiziert und wiedereingespielt werden können. Letzteres wird anhand einer sogenannten Kontrollstruktur, oder bookkeeping control structure (BCS), siehe Abbildung 7.7, realisiert, welche über die veränderten Seiten einer Anwendung Buch führt.

Die zugrundeliegende Datenstruktur der BCS entspricht einem sogenannten Radix-Baum, welcher in Linux zur effizienten Verwaltung der Page-Cache-Seiten verwendet wird. Jedes BCS-Baumblatt referenziert eine Seite und speichert die Bezeichnung jener inkrementellen Abbilddatei, welche die aktuellste Version der zugehörigen Seite enthält. Zusätzlich enthält es einen Dateioffset, da pro inkrementeller Abbilddatei mehrere Seiten abgespeichert werden können. Das Blatt wird durch Aufteilung der Seitenadresse in zwei, beziehungsweise drei Teile und anschließender Traversierung eines architekturabhängigen Tabellenpfades in *konstanter Zeit* erreicht.

Der BCS-Baum entspricht einem Metadatensatz, der bei jedem Checkpoint aktualisiert wird. Dabei werden BCS-Einträge, bisher unreferenzierter Seiten, hinzugefügt. Informationen existierender BCS-Einträge, die sich auf Seiten beziehen, die zwischen zwei inkrementellen Checkpoints verändert wurden, werden aktualisiert. Der BCS wird aktualisiert, falls, während des Checkpointings, Seitentabelleneinträge mit gesetztem Write-Bit gefunden werden.

Zum Ende einer Checkpointaktion wird der BCS persistent gesichert wird. Der BCS kann jedoch über den Checkpointvorgang hinaus im Speicher gehalten werden, sodass aus Performanzgründen vermieden werden kann, dass er beim nächsten Checkpoint erneut eingele-

sen werden muss. Unabhängig davon muss zu Beginn eines Restartvorgangs der BCS-Baum zunächst eingelesen werden.

Es reicht nicht aus, veränderte Speicherinhalte auf Seitenbasis zu erkennen, da hierdurch nicht alle Seitenänderungen erkannt werden. In diesem Zusammenhang werden veraltete BCS-Einträge nicht gelöscht, sodass falsche Daten beim Restart verwendet werden und eine Inkonsistenz entsteht. Beide Aspekte werden in den folgenden beiden Abschnitten behandelt.

7.5.4 Modifizierte Speicherregionen

In Linux werden *fortlaufende, virtuelle* Seiten mit gleichen Zugriffsrechten und gleichartigen Inhalten zu einer Speicherregion, beziehungsweise einer Virtual Memory Area (VMA) zusammengefasst. VMAs werden in einer Liste und in einem sogenannten *Red-Black-Tree* verwaltet. Anhand dieser Linuxstrukturen ist eine effiziente Prozessspeicherverwaltung möglich. Eine oder mehrere VMAs existieren pro Programmtext-, Halde-, Keller-, sowie den globale Variablen umfassenden Speicherbereich eines Prozesses. Jede, in den Adressraum eingeblendete Datei, beispielsweise die Standard-C-Bibliothek, als auch dynamische Speicheranforderungen (*malloc*), werden anhand einer eigenen VMA repräsentiert.

Zu einem Zeitpunkt gehört eine virtuelle Adresse zu einer VMA, zu unterschiedlichen Zeitpunkten hingegen kann eine virtuelle Adresse jeweils einer unterschiedlichen VMA zugeordnet werden, was in dem dynamischen Verhalten eines Prozesses begründet ist. Innerhalb eines Zeitintervalls kann beispielsweise

- eine neue VMA *erzeugt* werden, beispielsweise in Verbindung mit dynamischer Speicherallokation basierend auf den *mmap* und *brk*[10] Systemaufrufen,
- eine existierende VMA *vergrößert* wird, indem entweder angrenzende Adresslücken verwendet oder mit angrenzenden Speicherregionen, die übereinstimmende Zugriffsrechte besitzen, verschmolzen werden,
- eine existierende VMA *verkürzt* werden, indem *Adressintervalle am Anfang oder Ende der Region entfernt werden*,
- eine existierende VMA *in zwei aufgeteilt* werden, indem ein mittiges Adressintervall entfernt wird,
- eine existierende VMA entfernt werden.

Für das Verständnis der in Kapitel 7.5.5 aufgelisteten Fälle ist wichtig, dass Änderungen der VMA-Architektur eines Prozesses mit der BCS-Aktualisierung gekoppelt werden müssen. Falls das Write-Bit das einzige Kriterium ist, um eine Seitenänderung zu erkennen, kann

[10] Wird verwendet, um die Größe des Datensegments zu modifizieren.

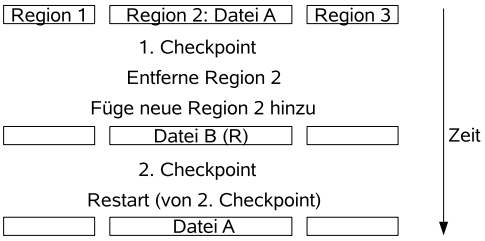

Abbildung 7.8: Austausch von Regionen gleicher Größe (Lesen)

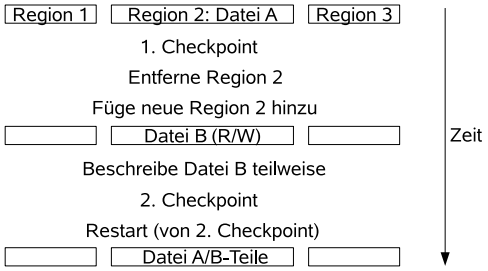

Abbildung 7.9: Austausch von Regionen gleicher Größe (Partiell beschreiben)

unbemerkt bleiben, dass vollständige Speicherregionen über die Zeit hin ausgetauscht werden. Systemverklemmungen oder -abstürze treten auf, wenn benötigte Seiteninhalte anhand veralteter BSC-Daten beim Restart nicht referenziert werden können. Inkonsistenzen entstehen, wenn Inhalte alter und neuer Regionen, den gleichen oder überlappenden Adressbereich betreffend, vermischt werden, neu entstandene Regionen jedoch unerkannt bleiben.

7.5.5 Klassifizierung veränderter Speicherregionen

Die in diesem Abschnitt aufgelisteten Fälle entstehen, wenn Speicherregionen, die in allen Fällen eine Untermenge gemeinsamer Seitenadressen besitzen, sich jedoch in Zugriffsrechten und Größe unterscheiden, sequentiell erzeugt, zerstört und erneut erzeugt werden.

7.5.5.1 Austausch von Speicherregionen gleicher Größe - Fall 1

Eine Anwendung blendet Datei A in die Speicherregion 2 ein. Später wird die Anwendung initial gesichert. Wird die Berechnung fortgeführt, wird die Datei ausgeblendet. An der Stelle der alten Region wird eine neue Region, gleicher Größe, erzeugt, welche Datei B einblendet. Wurde Datei B mit Leserechten eingeblendet und finden nur Lesezugriffe statt, wird das Write Bit-Flag der zugehörigen Seiten nicht gesetzt. Diese Seiten werden beim zweiten Checkpoint nicht berücksichtigt, was fehlerhaft ist. Bei Restart von Checkpoint 2 wird Datei A, anstelle von Datei B, eingeblendet, siehe Abbildung 7.8.

Wurde Datei B mit Schreibrechten eingeblendet und nur teilweise beschrieben, besitzt nur eine Untermenge der zu Region 2 gehörenden Seiten ein gesetztes Write-Bit. Der zweite Checkpoint resultiert darin, dass nur ein Teilbereich der Datei B abbildenden Seiten gesichert wird. Bei Restart vom zweiten Checkpoint wird eine Mischung aus Inhalten der Dateien A und B wiederhergestellt, was fehlerhaft ist, siehe Abbildung 7.9.

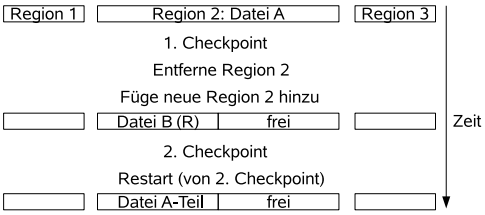

Abbildung 7.10: Austausch von Regionen unterschiedlicher Größe I (Lesen)

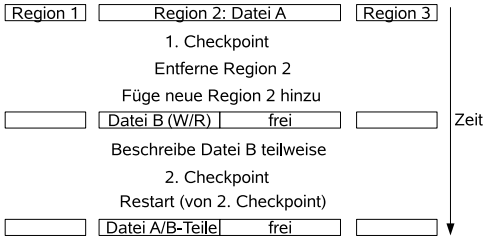

Abbildung 7.11: Austausch von Regionen unterschiedlicher Größe I (Partiell beschreiben)

7.5.5.2 Austausch von Speicherregionen unterschiedlicher Größe I - Fall 2

Dieses Szenario ist dem in Fall 1 beschriebenen sehr ähnlich. Im Unterschied zu Fall 1 wird Datei B (kleiner als Datei A) in eine kleinere zweite Region eingeblendet, sodass ein Bereich ungenutzer virtueller Adressen zwischen der neuen zweiten und bestehenden dritten Region nach dem ersten Checkpoint entsteht. Wurde Datei B mit Leserechten eingeblendet und finden nur Lesezugriffe statt, wird das Write-Bit zugehöriger Seiten nicht gesetzt, sie werden beim zweiten Checkpoint nicht berücksichtigt. Daher wird beim Restart von Checkpoint 2 die zweite Region mit Teilinhalten von Datei A initialisiert, was fehlerhaft ist, siehe Abbildung 7.10.

Wurde Datei B mit Schreibrechten eingeblendet und nur partiell beschrieben, besitzt nur eine Untermenge der zu Region 2 gehörenden Seiten ein gesetztes Write-Bit. Es wird ein Teilbereich der Datei B abbildenden Seiten beim zweiten Checkpoint gesichert. Bei einem Restart vom zweiten Checkpoint wird zwar die Struktur der neueren und kleineren Region 2 wiederhergestellt, jedoch stellt sie eine Mischung aus Inhalten der Dateien A und B dar, was fehlerhaft ist, siehe Abbildung 7.11.

7.5.5.3 Austausch von Speicherregionen unterschiedlicher Größe II - Fall 3

Dieses Szenario ist dem in Fall 1, siehe 7.5.5.1, beschriebenem sehr ähnlich. Im Unterschied zu Fall 1 wird Datei A (kleiner als Datei B) in eine kleinere zweite Region eingeblendet, sodass ein Bereich ungenutzter virtueller Adressen zwischen der zweiten und dritten Region vor dem ersten Checkpoint entsteht. Datei B wird nach dem ersten Checkpoint in die neue zweite Region eingeblendet, ohne dass ungenutzte Adressen zur dritten Region hin entstehen. Wurde Datei B mit Leserechten eingeblendet, wird kein Write-Bit bei den Datei B abbildenden Seiten gesetzt. In Folge werden beim zweiten Checkpoint keine Datei B beinhaltenden Seiten gesichert. Ein Restart auf Basis des zweiten Checkpoints stellt zwar die Struktur der neueren und größeren Region 2 wieder her, jedoch werden dessen Seiten mit Datei A Inhalten initialisiert. Da Datei A beziehungsweise die alte Region 2 kleiner als Datei B beziehungsweise die neue Region 2 ist, kann die Differenz an Seitenadressen

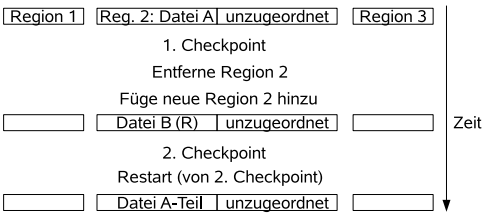

Abbildung 7.12: Austausch von Regionen unterschiedlicher Größe II (Lesen)

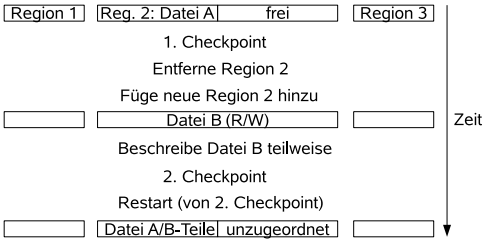

Abbildung 7.13: Austausch von Regionen unterschiedlicher Größe II (Partiell beschreiben)

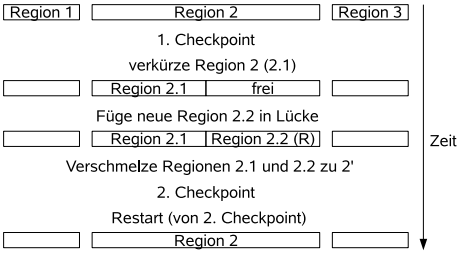

Abbildung 7.14: Verkürzte und vermischte Regionen (Lesen)

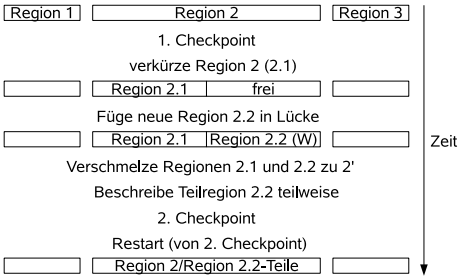

Abbildung 7.15: Verkürzte und vermischte Regionen (Partiell beschreiben)

nicht initialisiert werden, siehe Abbildung 7.12. Letzteres ist fehlerhaft und kann unter Umständen zu einem Systemabsturz führen.

Wurde Datei B mit Schreibrechten eingeblendet und nur teilweise beschrieben, besitzt nur eine Untermenge der zur Region 2 gehörenden Seiten ein gesetztes Write-Bit. Beim zweiten Checkpoint wird nur ein Teilbereich von Region 2 gesichert. Beim Restart vom zweiten Checkpoint werden Datei A- und Datei B-Inhalte miteinander vermischt. Andererseits kann die Differenz an Seitenadressen nicht initialisiert werden, siehe Abbildung 7.13.

7.5.5.4 Verkürzung/Vermischung von Speicherregionen - Fall 4

In diesem Szenario wird eine Region verkürzt, indem ein am Anfang, beziehungsweise Ende befindliches Adressintervall einer Speicherregion nach dem initialen Checkpoint entfernt wird. Vor dem zweiten Checkpoint wird eine neue Region erzeugt, welche das entfernte Adressinterval ausfüllt. Besitzen beide Regionen gleiche Zugriffsrechte, können sie sich zu einer zusammenschließen. Falls die neue und die zuvor verkürzte Region Leserechte besitzen, wird bei den Seiten der neuen Region nach dem ersten Checkpoint kein Write Bit-Flag gesetzt. Daher werden die Seiten der zweiten Region beim zweiten Checkpoint

ignoriert. Da es bisher kein Kriterium für zu löschende BCS-Einträge gibt (beispielsweise nach Entfernung von Adressintervallen), werden beim Restart vom zweiten Checkpoint Inhalte der alten, anstelle der neuen Region eingespielt, was fehlerhaft ist, siehe Abbildung 7.14.

Insofern die neue und zuvor die verkürzte Region Schreibrechte besitzen und nur partiell auf die neue Region geschrieben wird, werden beim zweiten Checkpoint nicht alle Seiten der neuen Region gesichert. Wie im vorherigen Teilszenario kann die Existenz veralteter BCS-Einträge beim Restart vom zweiten Checkpoint zur Vermischung von Inhalten der alten und neuen Region führen, was fehlerhaft ist, siehe Abbidlung 7.15.

7.5.6 Lösung: Speicherregionen-Monitor

Zusätzlich ist ein Speicherregionen-Monitor (SRM) notwendig, der das Speicherverhalten eines Prozesses überwacht und dieses Wissen für den BCS bereitstellt. Wenn ein BSC ausschließlich gültige Einträge enthält, werden alle unter 7.5.5 aufgelisteten Fehlerfälle vermieden.

Der SRM registriert, dass Speicherregionen erzeugt und entfernt worden sind, indem er zwischen zwei Checkpoints auftretende *mmap*, *munmap* und *brk* Systemaufrufe im Kern abfängt und jeweils eine geeignete BCS-Aktualisierungs-Operation ausführt. Das heißt, wurde *eine Speicherregion entfernt,* werden alle virtuellen Seiten ermittelt, welche sich im Intervall der Region befanden. In Folge können die zugehörigen Seiteneinträge der BCS ermittelt und gelöscht werden. Wurde *eine Speicherregion erzeugt,* wird die Start- und Endadresse der neuen Region im SRM vermerkt. Damit können insbesondere neue, mit Leserechten versehene, Seiten einer Region beim nächsten Checkpoint erkannt werden. Nach jedem Checkpoint werden alle SRM-Einträge gelöscht.

Die Erkennung von *mmap*, *munmap* und *brk*-Aufrufen wird anwendungstransparent im Kern vorgenommen. Die Kernfunktionen *do_mmap*, *do_munmap* und *do_brk* müssen hierzu minimal angepasst werden, damit der SRM kontaktiert wird, wenn *mmap*, *munmap* und *brk* von einem zu überwachenden Prozess aufgerufen werden[11].

Der SRM hat eine mit der BCS vergleichbare Struktur, siehe Abbildung 7.16. Ein 32-Bit-Zähler wird pro Monitor-Eintrag um einen Bit-Flagwert erhöht. Mithilfe des Zählers wird ein Pfad zum Baumknoten ermittelt, welcher die Start- und Endadresse der jeweiligen Region beinhaltet. Der Zähler wird am Ende jedes Checkpoints auf Null zurückgesetzt.

[11] Library Interposition wird an dieser Stelle nicht verwendet, weil abfangbare Benutzerebenenfunktionen keinen Zeiger auf die relevante VMA-Kernstruktur liefern können.

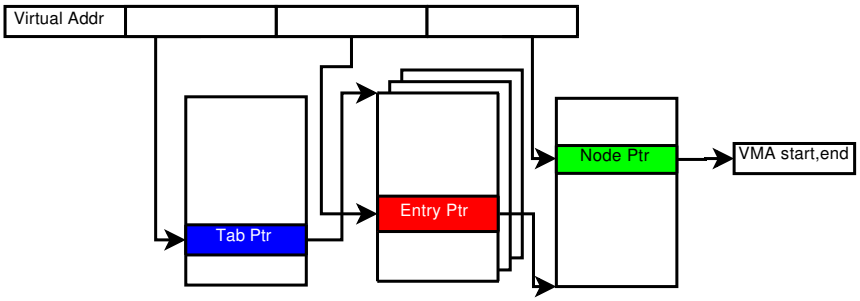

Abbildung 7.16: Speicherregionen-Monitor

7.6 Verwandte Arbeiten

In der Literatur wird adaptives Checkpointing überwiegend anhand eines variierenden Checkpointintervalls realisiert. In [114] werden geschätzte Netzwerk- und Knoten-Parameter, wie Knotenfehlerrate und Anzahl involvierter Knoten, als auch unter Verwendung des Checkpointaufwands, ungenutzter Rechenzeit und Latenz des Abbildtransfers, verwendet, um den Zeitpunkt des folgenden Checkpoints zu bestimmen. In [92] wird das optimale Checkpointintervall für Anwendungen im Hinblick auf das Recovery-Verhalten ermittelt.

Nach [30] wird die Checkpointingeffizienz gesteigert, wenn die Eigenschaften des verfügbaren Speichers berücksichtigt werden. In adaptiver Form werden hierbei zur Laufzeit Speichermedien mit den günstigsten Zugriffszeiten und unter Berücksichtigung der Abbildgröße ausgewählt.

Unter [95] hingegen rückt die Restarteffizienz in den Vordergrund. Sie wird gesteigert, wenn Abbilder beim Restart *schnell* verfügbar sind. Demnach selektiert ein Checkpoint-Replizierungsdienst (CRS) in adaptiver Weise potentielle Replikationsorte in Abhängigkeit des Verhältnisses Checkpointgröße zu Knotenspeicherkapazität, als auch der Netzwerkbandbreite.

Obwohl auch das jeweilige Checkpointingprotokoll Checkpoint- und Restartzeiten beeinflusst, nehmen die bisherigen Ansätze keinen Bezug dazu. In [35] wird lediglich blockierendes und nicht-blockierendes koordiniertes Checkpointing, in [23] koordiniertes und Nachrichtenaufzeichnung gegenübergestellt. Ein Wechsel zwischen verschiedenen Checkpointprotokollen wird nicht beschrieben. Des Weiteren wurde bisher überwiegend das System- und Netzwerkverhalten, jedoch nicht das Anwendungsverhalten einbezogen, um die Checkpointstrategie zu bestimmen.

Implementierungen inkrementellen Checkpointings können je nach Granularität veränderter Inhalte und deren Erkennung eingeteilt werden. Unter [64] werden veränderte Variablen mithilfe eines *manuell modifizierten* Compilers erkannt. In [148] hingegen werden inkrementell sichernde Zustandsvariable *automatisiert einer ausführbaren Datei hinzugefügt*. Diese Methode ist compilerunabhängig.

In [4] wird adaptives inkrementelles Checkpointing auf Basis einer sicheren Hashfunktion realisiert. Ändert sich der Hashwert eines Blocks zwischen zwei Checkpoints, liegt eine Veränderung vor. Mithilfe eines sogenannten *split-and-merge Algorithmus* kann das Speicherzugriffsverhalten der Anwendung aufgezeichnet werden, um aufzuteilende und zu vermischende Blocks zu bestimmen. Im Gegensatz zu bestehenden Ansätzen basierend auf Paging, bei denen geänderte Inhalte in statischen Größeneinheiten erfasst werden, lassen sich bei diesem selbst-optimierenden Ansatz dynamisch die Größen der sich geänderten Blöcke berechnen. Letzteres führt dazu, dass Abilddateigrößen um bis zu 25 Prozent reduziert werden.

Veränderte Inhalte auf Speicherseitengranularität werden unter Einbzug der MMU-Hardware und des Betriebssystems erkannt, siehe [63].

In [73] werden, nach Abänderung der *do_page_fault* Kernmethode das neunte und zehnte Bit eines Seitentabellen-Eintrages verwendet, um veränderte Seiten zu indizieren. In [132] wird, basierend der IA64-Architektur, ein neuer Systemaufruf vorgeschlagen, anhand dessen veränderte Seiten einer Anwendung von ihr selbst bestimmt werden können. Hierzu werden zwei zusätzliche Bits verwendet, die in den ignorierten Bits 53 und 54 eines Seitentabelleneintrags (Page Table Entry, PTE) gespeichert werden. Das sogenannte *syscall dirty bit* informiert darüber, ob eine Seite verändert wurde, nach dem letzten Lesen dieses Bits durch den Systemaufruf. Das sogenannte *kernel dirty bit* gibt Auskunft darüber, ob die Seite modifiziert wurde, nachdem der Kernel das native Dirty-Bit zurückgesetzt hat. Das native Dirty-Bit wird verwendet, um zu entscheiden, wann beide, das *kernel dirty bit* und das *syscall dirty bit*, gesetzt werden sollen. Das native Dirty-Bit wird jedoch nicht beeinträchtigt durch den neuen Systemaufruf. Dieser Ansatz ist IA64-spezifisch. Inwiefern die PTE-Bits 53 und 54 zukünftig für Checkpointing zur Verfügung stehen, ist nicht vorhersehbar. Zusätzlich müssen die Linux-Kernfunktionen *pte_dirty*, *pte_mkclean* und *pte_test_and_clear_dirty* modifiziert werden.

7.7 Zusammenfassung

Adaptives Checkpointing zielt auf einen Effizienzgewinn gegenüber herkömmlichem Checkpointing ab, indem das Checkpointingverhalten für eine Anwendung dynamisch angepasst wird. Hierbei wird das System- und Anwendungsverhalten einbezogen, um eine optimale Checkpointingstrategie zu identifizieren.

In der Literatur wurden bisher verschiedenene Checkpointingprotokolle separat, jedoch nicht deren Übergänge zueinander analysiert. In diesem Kapitel wurde erstmals der beidseitige Wechsel zwischen koordiniertem und unkoordiniertem Checkpointing betrachtet, wobei die Anwendungsausführung nicht unterbrochen werden muss. Dieser Wechsel ist von großem Vorteil, wenn die Anzahl an Anwendungsprozessen stark variiert und Prozesssynchronisierungen, aus Effizienzgründen, nicht zu jedem Zeitpunkt durchgeführt werden können. Aufgrund einer fehlenden globalen Zeit und unterschiedlicher Nachrichtenlaufzeiten tritt der Impuls zum Protokollwechsel nicht bei allen Einheiten einer verteilten Anwendung zum selben Zeitpunkt ein. Die entstehenden Sonderfälle wurden analysiert und Lösungswege dargelegt, um Inkonsistenzen und den Verlust wichtiger Zwischenzustände zu vermeiden. Die Notwendigkeit eine protokollübergreifenden Abbildverwaltung wurde bis dato nicht betrachtet.

Inkrementelles Checkpointing kann Checkpointingaufwand, insbesondere Abspeicherungszeit und Abbildgröße, entscheidend verringern, abhängig vom Schreibzugriffsmuster der Anwendung. Mit dem beidseitigen Wechsel zwischen inkrementellem und vollständigem Sichern kann eine weitere Facette adaptiven Checkpointings realisiert werden. In diesem Kapitel wurden die Vor- und Nachteile inkrementellen Checkpointings, basierend auf dem Dirty-Bit- und dem Write-Bit-Ansatz, ausführlich erläutert. Der Dirty-Bit-Ansatz ist nicht gangbar. Beim derzeitigen Linux-Entwicklungsstand existieren jedoch auch Einschränkungen beim Write-Bit-Ansatz. Erstmals wurde dargelegt, dass alle Speicherinhaltsänderungen beim Dirty- oder Write-Bit-Ansatz nur erkannt werden, wenn zusätzlich zur Überwachung einzelner Seiten auch vollständige Speicherregionen auf Veränderungen hin kontrolliert werden. Die entstehenden Sonderfälle wurden analysiert, ein Lösungsweg wurde aufgezeigt und implementiert. Die entsprechenden Messergebnisse werden in Kapitel 8.5 präsentiert. Das Schreibzugriffsmuster einer Anwendung wurde mit einem neuartigen, anwendungstransparenten Monitor im Kernel überwacht. Mithilfe dieser Monitorinformationen kann adaptiv zwischen vollständigem und inkrementellem Checkpointing umgeschalten werden.

8 Messungen und Bewertung

Dieses Kapitel untersucht die Effizienz und Tauglichkeit bedeutender GCA-Komponenten. In diesem Zusammenhang wird ermittelt, wie groß der GCA-Aufwand einer Sicherung und Wiederherstellung ist, gegenüber der Verwendung nativer Checkpointer. Andererseits wurde in der GCA koordiniertes, unkoordiniertes und inkrementelles Checkpointing unter Einbezug heterogener Checkpointer in der GCA implementiert, um adaptives Checkpointing zu ermöglichen. Das Fehlertoleranzverhalten dieser Implementierungen wird anhand sequentieller und verteilter Anwendungen untersucht.

Eine Fehlertoleranzaktion wird maßgeblich davon beeinflusst, wie schnell Checkpointabbilder geschrieben und gelesen werden können. Hierbei spielen zugrundeliegende Dateisysteme, als auch verwendete Speicher- und Netzwerk-Hardware eine wichtige Rolle. Dieser Zusammenhang wird in Kapitel 8.4 genauer betrachtet.

In diesem Kapitel werden überwiegend die *Gesamtkosten* von Checkpoint/Restart ausgewiesen, anstelle auf Zeiten von Teilsequenzen, beispielsweise der Synchronisierung, zu reduzieren, wie es vereinzelt in der Checkpointingliteratur praktiziert wird.

8.1 Messumgebung

Als Messplattform wird ein Cluster, bestehend aus 16 Knoten, eingesetzt, in dem ein Preboot Execution Environment (PXE)-Server als Startumgebung für die Client-Knoten dient. Der Server besitzt eine Intel Celeron CPU 420 mit 1,6 GHz Taktgeschwindigkeit und 1 GB Hauptspeicher. Eine Festplatte vom Typ Samsung SpinPoint S166 HD161HJ (SATA 300) mit 7200 U/Min, 4,17 ms durchschnittlicher Spursuche- und Sektorpositionierungszeit und 8 MB Cache ist integriert. Die Lese-Geschwindigkeit beträgt maximal 120 MB/s, geschrieben wird mit maximal 300 MB/s. Jeder Client-Rechner besitzt zwei AMD Opteron Prozessoren 244 (Ein-Kern-CPU), jeweils mit einer Taktgeschwindigkeit von 1,8 GHz und 2 GB Hauptspeicher. Eine Festplatte vom Typ Maxtor 6Y080L0 (ATA 133) mit 7200 U/-Min, 9 ms durchschnittlicher Spursuche- und Sektorpositionierungszeit und 2 MB Cache ist integriert. Nach [39] besitzt sie eine Schreibgeschwindigkeit von 26,54 MB/s und Lesegeschwindigkeit von 47,95 MB/s. Der PXE-Server und Client-Rechner sind über Gigabit-Netzwerkleitungen miteinander verbunden. Optional zur Nutzung der Clientfestplatte fungiert der PXE-Server gleichzeitig als NFS-Server, das heißt, Clients legen Daten persistent auf der Festplatte des NFS-Server-Rechners ab, beziehungsweise lesen Daten von dort ein.

Beide Festplatten sind mit dem ext3-Dateisystem formatiert.

Weil keine Benchmarks für Checkpointing existieren, wurde eine in der Speichergröße skalierbare (10, 50, 100, 500, 1000 MB), generische Testanwendung[1] (P10, P50, P100, P500, P1000) verwendet. Diese Testanwendung wird sequentiell (ein Prozess) oder verteilt (ein Prozess pro Gridknoten) ausgeführt und alloziert dynamisch Speicher im Heap. Mit steigender Speichergröße wird eine verlängerte Checkpoint-/Restart-Dauer erwartet.

Unkoordiniertes Checkpointing und die Gridkanalsicherung, siehe Kapitel 6, werden anhand einer gesonderten Client-Server-Anwendung, mit variabler Nachrichtengröße und variabler Sendefrequenz, untersucht, siehe Kapitel 8.6 und 8.7.

8.2 Native Checkpointer versus GCA

In den Abbildungen 8.1 und 8.2 wird gegenübergestellt, wieviel Zeit einerseits von nativem[2] BLCR, MTCP, SSI und andererseits von der GCA benötigt wird, um eine Job-Einheit mit BLCR, MTCP und SSI zu sichern. Die Abbilder werden dabei unter NFS abgelegt. Die Abbildgröße 10 MB bezieht auf den Checkpoint der Testanwendung P10, Abbildgröße 50 MB auf P50 und so weiter. Alle Messungen wurden mehrfach wiederholt. Neben Durchschnittswerten werden Fehlervarianzen anhand von Minimal- und Maximalwerten angegeben.

Wie erwartet steigt die Sicherungsdauer mit zunehmender Anwendungsgröße in den Diagrammen beider Abbildungen. Bei den Testanwendungen P10 bis einschließlich P500 ergibt sich für die nativen Checkpointern BLCR, MTCP und SSI gegenüber der GCA-initiierenden Sicherung ein Geschwindigkeitsvorteil, die Differenz liegt jedoch im Millisekundenbereich. Bei P1000 treten Schwankungen im Sekundenbereich auf. Hierbei ist natives MTCP und natives SSI mehrere Sekunden schneller als GCA-MTCP und GCA-SSI. Hingegen ist natives BLCR ca. 2 Sekunden langsamer als GCA-BLCR. Diese Schwankungen sind einerseits auf die Dynamiken bei der Netzwerkübertragung zurückzuführen. Andererseits führt die sich ständige ändernde Anordnung freier und belegter Blöcke bei mehreren Schreibvorgängen zu unterschiedlichen Positionierungszeiten des Festplattenschreibkopfes.

Insgesamt betrachtet existiert ein vernachlässigbarer GCA-Aufwand beim Checkpointing gegenüber den nativen Checkpointern. Dieser ist darauf zurückzuführen, dass die Sequenzen *prepare*, *stop*, *checkpoint* und *resume* jedes Checkpointers explizit von der Übersetzungsbibliothek angesteuert und Grid-Checkpointing Metadaten angelegt werden müssen.

Die Abbildungen 8.3, 8.4, 8.5 und 8.6 stellen die Zeiten der GCA-Sequenzen *prepare*, *stop checkpoint* und *resume* Phase für die Testanwendungen P10 und P1000 dar. Die *prepare*-,

[1] In der Literatur werden Checkpointingprotokolle häufig mithilfe der NASA NAS Benchmarks evaluiert, welche Testanwendungen mit verschiedenen Kommunikationsschemen enthält. Der Ein-/Ausgabe-Einfluss großer Abbildern kann mit diesen Anwendungen jedoch nicht evaluiert werden.

[2] BLCR, MTCP und SSI werden unverändert, ohne die für die Integration in die GCA notwendigen Modifikationen, verwendet.

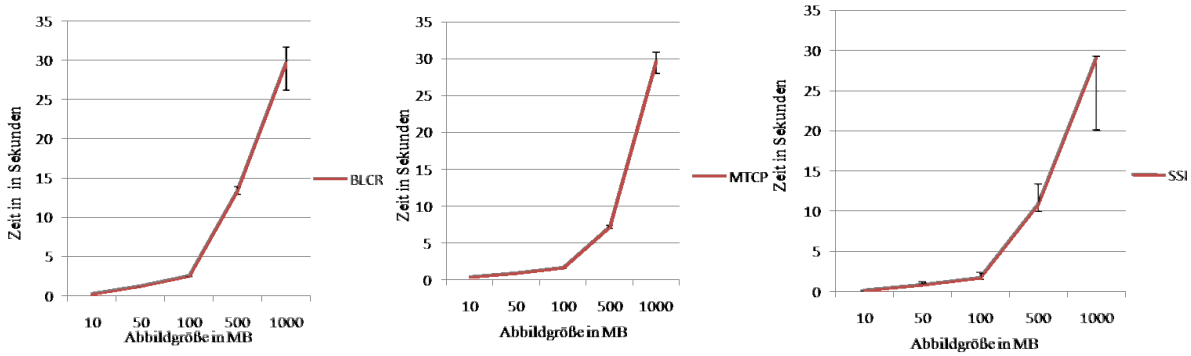

Abbildung 8.1: Checkpointing mit nativem BLCR (l), MTCP (M) und SSI (r)

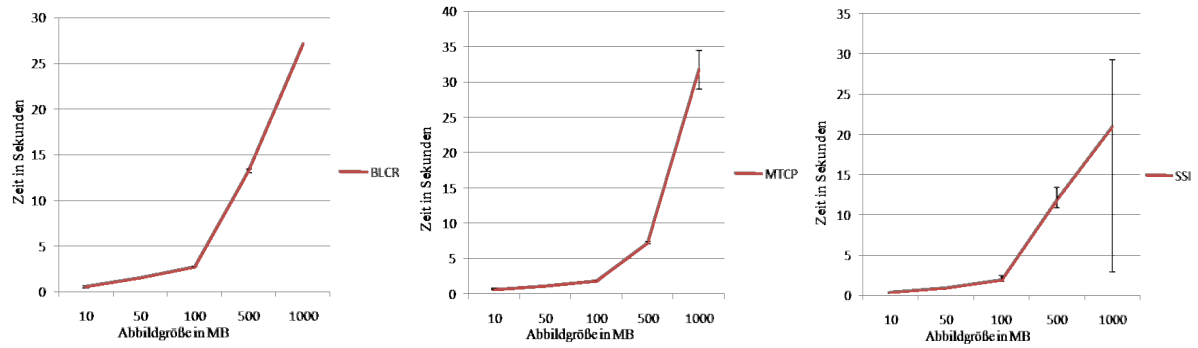

Abbildung 8.2: GCA Checkpointing mit modifiziertem BLCR (l), MTCP (M) und SSI (r)

stop- und *resume*-Phasen benötigen jeweils bei heterogenen Checkpointern und bei zwei Testanwendungen mit stark variierender Speichergröße (10 MB und 1000 MB) nahezu eine konstante Zeitdauer. Im Gegensatz hierzu wächst die Dauer der *checkpoint*-Phase mit der Speichergröße der Anwendung, wie erwartet. Das ist hier insbesondere auf das Eingabeverhalten der zugrunde liegenden Festplatte und das Dateisystem zurückzuführen, siehe Kapitel 8.4. Weil bei den nativen Checkpointern nicht mehrere Sequenzen über eine Übersetzungsbibliothek angesteuert werden müssen und entsprechende Barrieren in der nativen Checkpoint-Sequenz vorgenommen werden müssen, entsteht ein vernachlässigbarer GCA-Aufwand, der jedoch gleichförmig, da unabhängig von der Abbildgröße, ist.

Die bisherigen GCA-Daten entsprechen der Sicherungs- und Wiederherstellungsdauer einer Job-Einheit und dienen damit gleichzeitig als Messdaten für unkoordiniertes Checkpointing. Die Messdaten zur Recovery-Line-Berechnung werden in Kapitel 8.6 diskutiert.

In den Abbildungen 8.7 und 8.8 wird der BLCR-, MTCP- und SSI-native Restart dem GCA-initiiertem Restart gegenübergestellt. Während bei den Testanwendungen P10, P50 und P100 die Restartdifferenz im Millisekundenbereich liegt, sind es insbesondere bei P500 (BLCR), P1000 (MTCP) und P500 (SSI) mehrere Sekunden.

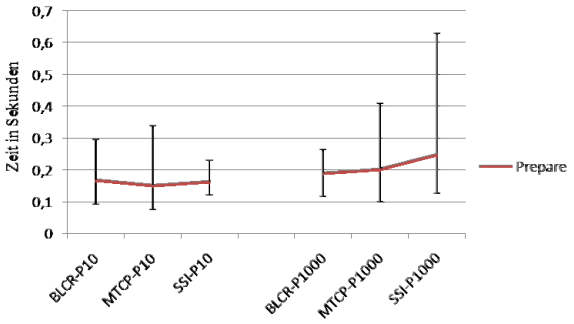

Abbildung 8.3: Prepare-Phase

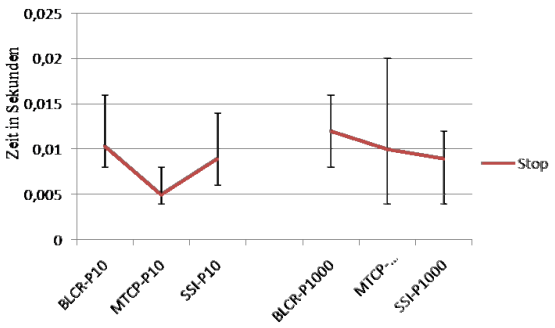

Abbildung 8.4: Stop-Phase

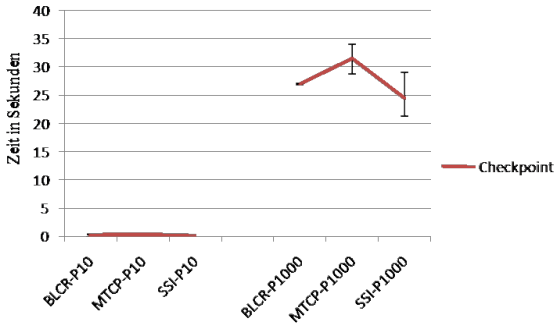

Abbildung 8.5: Checkpoint-Phase

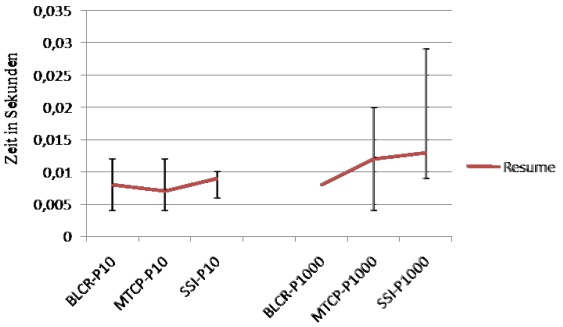

Abbildung 8.6: Resume-Phase

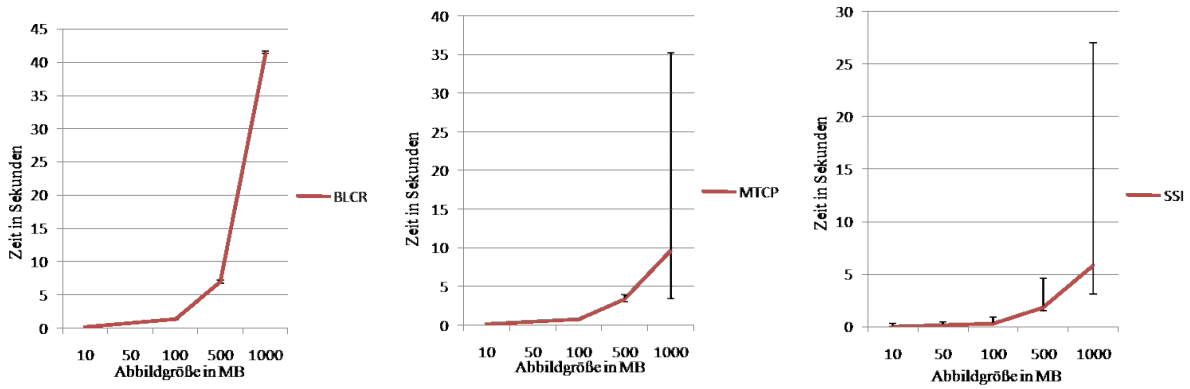

Abbildung 8.7: Restart mit nativem BLCR (l), MTCP (M) und SSI (r)

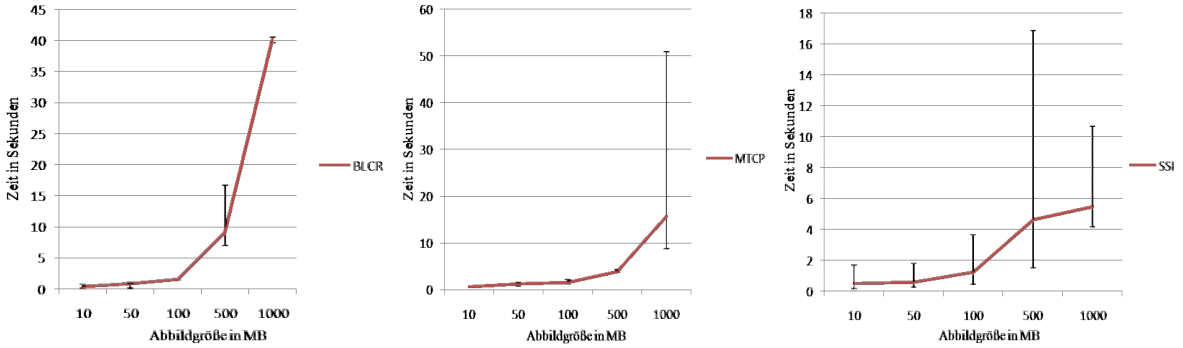

Abbildung 8.8: GCA Restart mit BLCR (l), MTCP (M) und SSI (r)

Insgesamt betrachtet verändert die GCA-Übersetzungsbibliothek die Leistungsverhältnisse der involvierten Checkpointer jedoch nicht. Analog zu oben entsteht der Aufwand dadurch, dass der native Restartaufruf in die Phasen *rebuild* und *resume* aufgeteilt werden musste, um einen koordinierten Restart zu ermöglichen und Grid-Checkpointing Metadaten eingelesen werden müssen. Die Schwankungen sind hierbei auf das Dateisystem zurückzuführen, siehe Kapitel 8.4.

8.3 Die GCA und verteiltes Checkpointing

Die Abbildungen 8.9 und 8.10 veranschaulichen koordiniertes Checkpointing einer *verteilten* Anwendung, beziehungsweise eines Jobs mit mehreren Job-Einheiten, Kommunikationskanäle werden hierbei nicht berücksichtigt. Bei der Datenerhebung wurden zwei unterschiedliche Knoten-Konstellationen verwendet. *SSI-MAJ* entspricht neun logischen

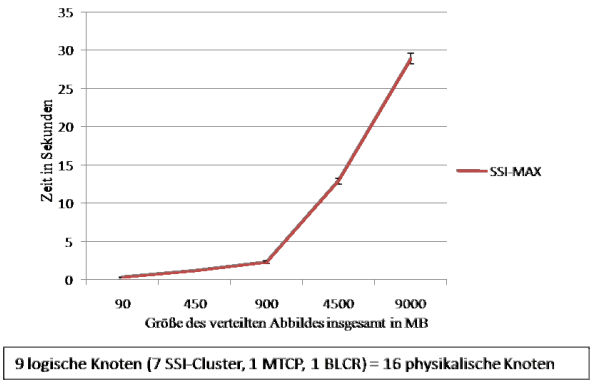

Abbildung 8.9: SSI-MAJ Checkpoint

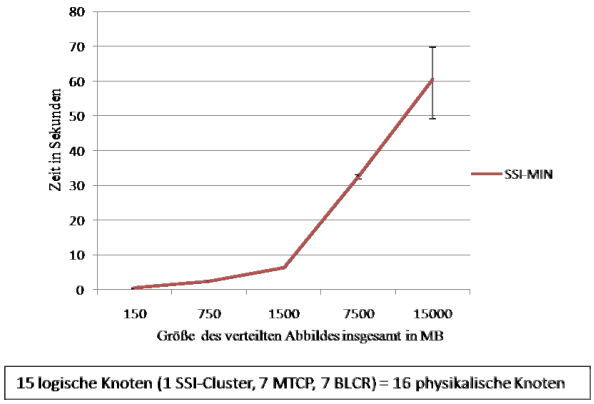

Abbildung 8.10: SSI-MIN Checkpoint

Knoten: sieben SSI-Clustern[3], einem Knoten mit BLCR und einem mit MTCP. Hingegen besteht *SSI-MIN* aus 15 logischen Knoten: einem SSI-Cluster, sieben Knoten mit BLCR und sieben Knoten mit MTCP. Hierdurch sollen unterschiedliche Checkpointergewichtungen simuliert werden.

Aufgrund der höheren Knotenanzahl bei SSI-MIN (15 logische Knoten) und dem daraus resultierenden Speicheraufwand sind alle Checkpointingzeiten von SSI-MAJ (9 logische Knoten) niedriger, weil weniger geschrieben werden muss. Während bei SSI-MIN 60s benötigt werden, um 15 mal 1 GB zu sichern, sind es bei SSI-MAJ hingegen 30s für 9 mal 1 GB. Weil SSI-MIN, trotz des logischen Knotenverhältnisses von 9:15, doppelt so lange wie SSI-MAJ benötigt, wird klar ersichtlich, dass der SSI Checkpointer effizienter sichert als MTCP und BLCR. Weiterhin beeinflusst der bei 15 Job-Einheiten höhere NFS-Übertragungs- und Speicheraufwand die Checkpointdauer im Vergleich zu neun oder einer Job-Einheit, siehe Kapitel 8.2.

In den Abbildungen 8.11 und 8.12 werden die Wiederherstellungszeiten von SSI-MAJ und SSI-MIN angegeben. Hierbei fallen die deutlich höheren Restart-Laufzeiten von P500, respektive 4500 MB auf der x-Achse bei SSI-MAJ und 7500 MB auf der x-Achse bei SSI-MIN und P1000, respektive 4500 MB bei SSI-MAJ und 15000 MB bei SSI-MIN auf. Für die Wiederherstellung des 9000 MB großen Abbilds werden bei SSI-MAJ 630s, für das 15000 MB große Abbild bei SSI-MIN hingegen 2350s benötigt. Im folgenden Kapitel 8.4 werden die für diese Werte zugrundeliegenden Zusammenhänge näher beleuchtet.

[3]Ein minimales SSI-Cluster besteht im Testaufbau aus *zwei* physikalischen Knoten, die als ein logischer Knoten betrachtet werden.

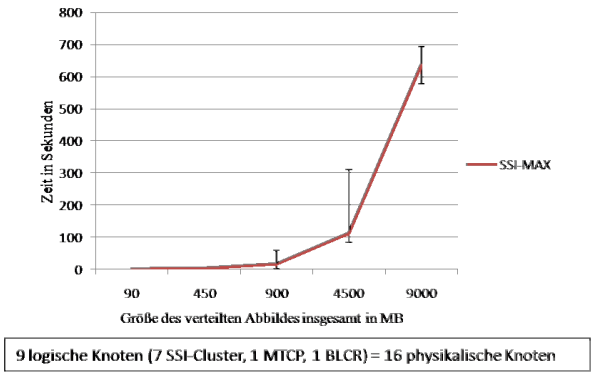

Abbildung 8.11: SSI-MAJ Restart

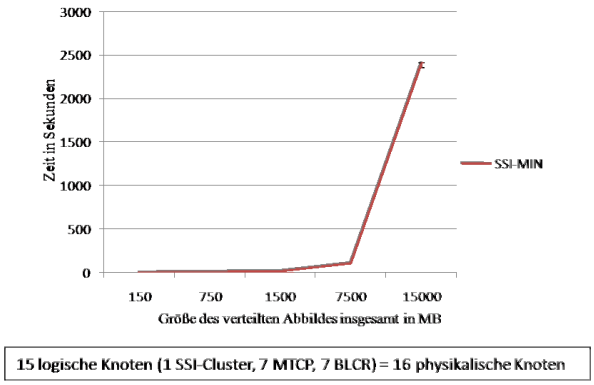

Abbildung 8.12: SSI-MIN Restart

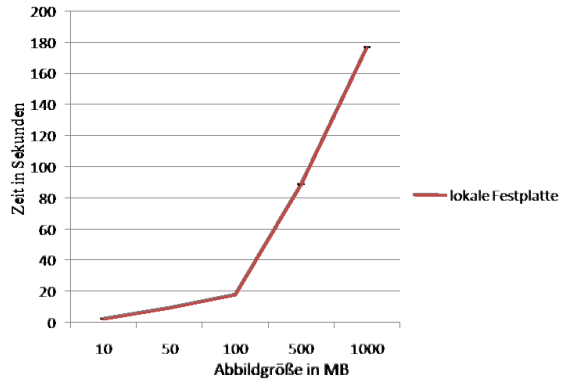

Abbildung 8.13: Checkpoint auf lokale Festplatte schreiben

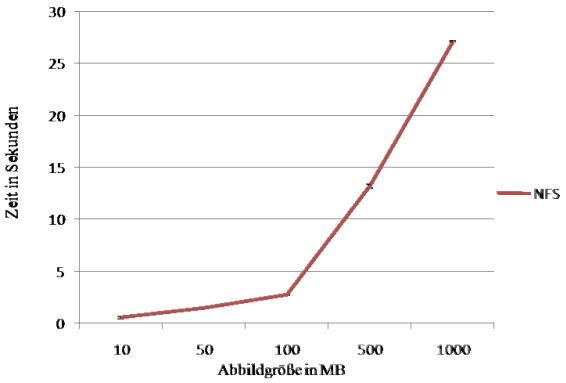

Abbildung 8.14: Checkpoint auf NFS Volume schreiben

8.4 Ein-/Ausgabe

8.4.1 Lokale Festplatte versus NFS

Tabelle 8.1: Das Abbild wird auf die lokale Festplatte und auf NFS geschrieben

	Lokale Festplatte	NFS
500 MB schreiben	83s	13s
1000 MB schreiben	177s	30s

Checkpointing wird überwiegend von den Eigenschaften zugrundeliegender, heterogener Speichertechnologien (Hard- und Software) bestimmt, vor allem vom Datendurchsatz, der Latenz und den Zugriffszeiten. Dieser Aspekt wird in den Abbildungen 8.13 und 8.14 verdeutlicht. Hierbei werden die Zeiten angeben, um die Abbilder der Testanwendungen P10,

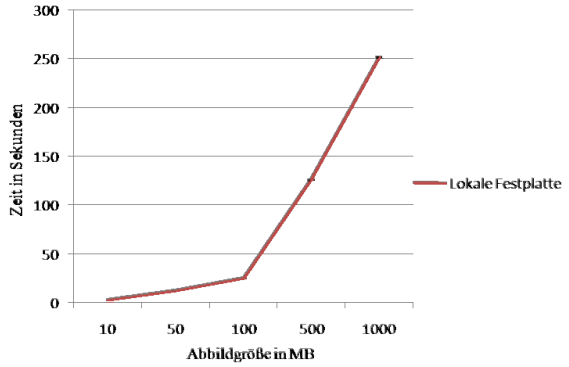

Abbildung 8.15: Checkpoint von lokaler Festplatte lesen

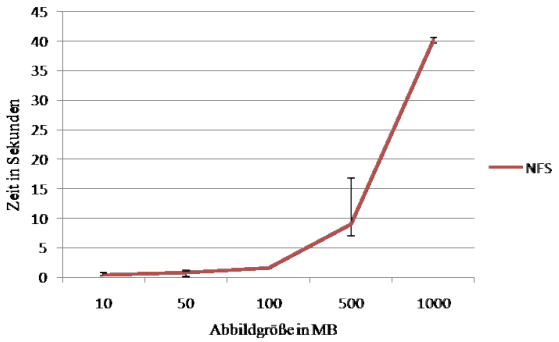

Abbildung 8.16: Checkpoint von NFS Volume lesen

Tabelle 8.2: Das Abbild wird von der lokalen Festplatte und von NFS eingelesen

	Lokale Festplatte	NFS
500 MB lesen	125s	9s
1000 MB lesen	250s	40s

P50, P100, P500 und P1000 (jeweils eine Job-Einheit) auf der lokalen Festplatte und unter NFS[4] zu sichern. Die Tabelle 8.1 hebt relevante Messdaten hervor, aus denen ein deutlicher Performanzvorteil NFSs gegenüber der lokalen Festplatte beim NFS-Client hervorgeht. Werden die Durchschnittswerte gegenübergestellt, ergibt sich für P500 ein Verhältnis von 6,64:1 (88s und 13s) sowie 6,03:1 (177s und 30s) für P1000. Obwohl die NFS-Serverfestplatte mit 300 MB/s beschrieben werden kann, sind es aus NFS-Clientsicht deutlich weniger. Dies liegt einerseits an der Netzwerkbandbreite von 134,22 MB/s (1Gbit/s) begründet. Demnach können weniger als die Hälfte der Daten transportiert werden, welche die NFS-Serverfestplatte idealerweise schreiben kann. Die Netzwerkbandbreite fungiert daher als Performanz-Flaschenhals. Andererseits wird die optimale Schreibgeschwindigkeit nicht erreicht, weil offensichtlich nicht genügend freie und *aneinanderliegende* Festplattensektoren zur Verfügung stehen. Die Performanz wird daher zusätzlich von der individuellen Spurpositionierungs- und Sektorsuchzeit beeinflusst. Der Geschwindigkeitsvorteil von NFS liegt zudem darin begründet, dass asynchron geschrieben wird, siehe Kapitel 8.4.2.

Analog zu oben stellen die Abbildungen 8.15 und 8.16 die zugehörigen Abbild-Einlesezeiten dar. Auch hier besitzt NFS die höhere Performanz. Weil die Lesegeschwindigkeit der NFS-Serverfestplatte ungefähr mit der Netzwerkbandbreite übereinstimmt, ist das Einlesen von 500 MB in 9s und 1000 MB ins 40s nachzuvollziehen, wenn die zwischenliegenden Puffer berücksichtigt werden. Hingegen entstehen von der Festplattenspezifikation stark abwei-

[4] In der NFS-Konstellation werden Abbilder vom Hauptspeicher des NFS-Clients über den Systembus und das Gigabit-Netzwerk zum NFS-Server gesendet, welcher eintreffende Daten puffert und unter dem *ext3*-Dateisystem des lokalen Festplattenspeichers ablegt.

chende Werte, wenn die beiden Abbilder von der lokalen Festplatte eingelesen werden, siehe Tabelle 8.2. Anstelle 50 MB/s einzulesen, werden für 500 MB und 1000 MB jeweils mehr als zehn Mal so viel benötigt. Dies kann nur mit dem vierfach kleineren Cache und dem Such- und Lesekopf-Positionierungsaufwand erklärt werden, der entsteht, wenn zu lesende Speicherblöcke nicht fortlaufend zusammen hängen.

8.4.2 Asynchrones Schreiben bei NFS

Mit NFS können Daten asynchron geschrieben werden, was zu dem unter 8.4.1 erwähnten NFS-Geschwindigkeitsvorteil beiträgt. Demnach puffert der NFS-Server Schreibzugriffe des NFS-Clients und übergibt die Daten dem zugrundeliegenden lokalen Dateisystem, ohne darauf warten zu müssen, dass alle Daten auf stabilem Speicher festgeschrieben worden sind. Es besteht keine Notwendigkeit eines expliziten *sync*-Aufrufs[5], um sicher zu gehen, dass alle Dateien auf Festplatte geschrieben wurden sind. Der mit *sync* verbundene Aufwand, welcher von Festplattenlatenzen abhängt und mit wachsenden Abbildern steigt, wird vermieden.

8.4.3 Verteiltes Checkpointing und NFS-Ein/Ausgabe

Tabelle 8.3: Fehlertoleranzzeiten bei steigender Anzahl Job-Einheiten

Job-Einheiten	1	9 (SSI-MAJ)	15 (SSI-MIN)
Checkpoint	13s (500 MB)	13s (4500 MB)	32s (7500 MB)
Checkpoint	27s (1000 MB)	28s (9000 MB)	60s (15000 MB)
Restart	9s (500 MB)	111s (4500 MB)	110s (7500 MB)
Restart	40s (1000 MB)	635s (9000 MB)	2409s (15000 MB)

Mit einer steigenden Dateisystembelastung reduziert sich jedoch auch der Vorteil asynchronen Schreibens. In Tabelle 8.3 wird dargelegt, wie sich die NFS-Speicherperformanz verschlechtert, wenn ein oder mehrere Job-Einheiten gleichzeitig verschiedene Abbilder schreiben, als auch einlesen. Demnach können eine sowie neun Job-Einheiten vom NFS-Server noch ungefähr im gleichen Zeitumfang gesichert werden. Hingegen benötigt er bei 15 Job-Einheiten mehr Zeit. Dies ist darauf zurückzuführen, dass bei mehreren eingehenden Datenströmen auf der NFS-Serverseite mehr geschrieben werden soll, als die *Festplattenbandbreite* ermöglicht. Hierdurch wird ersichtlich, dass die *Netzwerkbandbreite* nicht mehr der alleinige Flaschenhals ist, wie beispielsweise bei der Sicherung einer Job-Einheit, siehe Kapitel 8.2.

[5] Im Page Cache befindliche Daten werden unter Linux erst dann mit dem Gegenstück auf Platte synchronisiert, wenn ein bestimmter Schwellenwert zu synchronisierender Daten erreicht, oder ein Zeitintervall überschritten wurde.

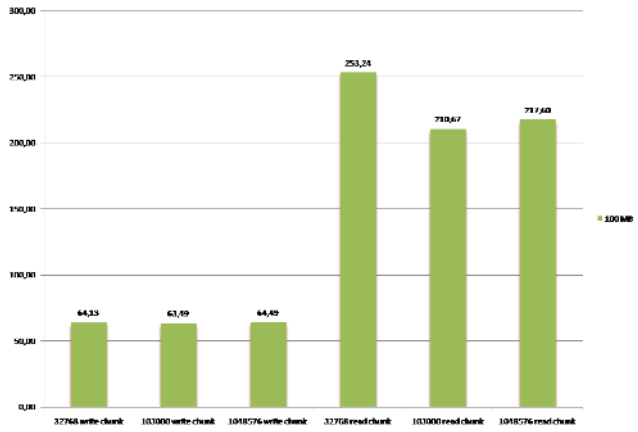

Abbildung 8.17: NFS und verschiedene Blockgrößen

Der Einfluss der Festplattenbandbreite ist besonders auffällig, wenn die Job-Einheiten wiederhergestellt werden. Hierbei treten starke Sprünge auf. Während bei 1000 MB 40s benötigt werden, sind es bei 9000 MB 635s und 2409s bei 15 GB. Um diese Werte eindeutig zu erklären, muss genau aufgeschlüsselt werden, wieviel Zeit der NFS-Server, die Datenübertragung und die Speicherallozierung und Speicherreinitialisierung auf der Clientseite benötigen. Dies ist jedoch nur möglich, wenn Checkpointer und Dateisystem entsprechend modifiziert werden. Im Rahmen dieser Arbeit ist dies nicht möglich.

Aber auch Blockgrößen auf NFS- und Festplattenebene spielen eine entscheidende Rolle bei nebenläufigen Speicherzugriffen. Je größer die Blockgrößen, desto weniger teure Adressierungsoperationen müssen durchgeführt werden. Bei einem verteilten Dateisystem kann der Datenzugriffsaufwand optimiert werden, indem die *optimale Blockgröße für eine spezifische Umgebung* bestimmt wird. Abbildung 8.17 setzt drei unterschiedliche NFS-Blockgrößen und die mit ihnen verbundenen Lese- und Schreibzeiten eines 100 MB großen Abbilds in Bezug zueinander. Obwohl sich die Schreibzeiten nicht wesentlich unterscheiden, machen sich unterschiedliche Größen beim Lesen bemerkbar. Dabei gilt, dass die größte Blockgröße nicht zwangsläufig gleich die optimale darstellt. NFS ermittelt beispielsweise automatisch die am besten geeignetste zur Laufzeit.[6]

Ein weiterer Faktor für variierende Einlesezeiten bildet der Umstand, ob ein *warmer* oder *kalter* Cache vorliegt. Ein warmer Cache kann vollständige Dateien, beziehungsweise Dateifragemente vorhalten, sodass teure Plattenzugriffe vermieden oder verringert werden. Zugriffszeiten beim Einlesen werden hierdurch reduziert. In [86] wird bei warmen Caches ein Performanzgewinn von 0.5s erzielt. Bei Restart nach Knoten-Reboot gelten immer die Werte eines kalten Caches.

[6]Trotz eines, durch variierende NFS-BLockgrößen erzielten, Performanzgewinns können die zugrundeliegenden Begrenzungen der Festplatten-Bandbreite nicht aufgelöst werden.

Um den mit NFS-verbundenen Nachteil des Server-Bottlenecks zu vermeiden empfiehlt es sich, andere verteilte Dateisysteme zu nutzen. Bei XtreemFS [77] werden beispielsweise Blöcke einer Datei auf *mehrere* Knoten verteilt (engl. striping), sodass eine höhere Zugriffsgeschwindigkeit entsteht.

8.4.4 Speicheralternativen

Alternativ zu herkömmlichem magnetischen Festplatten-Speicher (MFS) können Festkörperlaufwerke (engl. solid state disk, SSD) eingesetzt werden. Weil hierbei viele kleine Flash-Speicher-Controller integriert werden, auf die *nebenläufig zugegriffen* werden kann, steigt die Lese- und Schreibperformanz gegenüber Magnet-Speichern in zunehmendem Maße beträchtlich, [62]. Letzteres ist die Basis, asymmetrisches Schreib- und Leseverhalten mit SSS's realisieren zu können.[7] Die Messungen der Autoren belegen, dass zwar das Schreibverhalten von MFS und SSD einander gleicht, jedoch SSD sich über *alle* Sektoren hinweg konstant verhält.[8] Die Performanz des zufälligen Einlesens ist bei SSD's deutlich höher. Hieraus ergibt sich, mit Blick auf die in Tabelle 8.3 dargestellten Wiederherstellungszeiten, das Potential, Restart-Zeiten entscheidend zu verringern.

8.5 GCA - inkrementelles Checkpointing

Neben den Eigenschaften diverser Speichertechnologien kann der Ein-/Ausgabe-Aufwand direkt auf der Ebene des Checkpointing-Strategie reduziert werden. Inkrementelles Checkpointing kann, abhängig vom Schreibzugriffsmuster der Anwendung und des Checkpointingintervalls, Abbildgrößen reduzieren und damit Checkpointingzeiten verkürzen.

Die Anwendungen P10, P50, P100, P500 und P1000 aus Kapitel 8.1 werden für die Messungen modifiziert. Jetzt beschreiben sie, in aufsteigender Richtung und wiederholter Form, seitenweise den eigenen Adressraum in Abständen von 100ms. In Abbildung 8.18 sind die durchschnittlichen *Checkpointingzeiten* angegeben, die ein SSI-Checkpointer benötigt, um P10, P50, P100, P500 und P1000 regulär und inkrementell zu sichern. Die Durchschnittswerte basieren pro Anwendung auf zehn Messungen, wobei jeder einzelne Checkpoint drei Sekunden nach Beendigung des vorangehenden Checkpoints erzeugt wurde.

Es ist ersichtlich, dass inkrementelles Checkpointing bei allen Testanwendungen schneller ist. Dies ist auf den reduzierten Schreibaufwand zurückzuführen, da nur die zwischenzeitlich veränderte Seiten gesichert werden müssen. Obwohl anwendungsübergreifend in den gleichen Zeitintervallen gesichert wurde, wächst die Sicherungszeit mit dem Umfang

[7] MFS realisiert optimalen Zugriff, wenn zusammenhängende große Blöcke und geringe Positionierungseiten vorliegen.

[8] Der physikalische Aufbau einer MFS verursacht eine Varianz der Übertragungsrate bei innereren und äußeren Spuren. Dies tritt bei SSD nicht auf, da es keine rotierenden, mechanischen Elemente gibt.

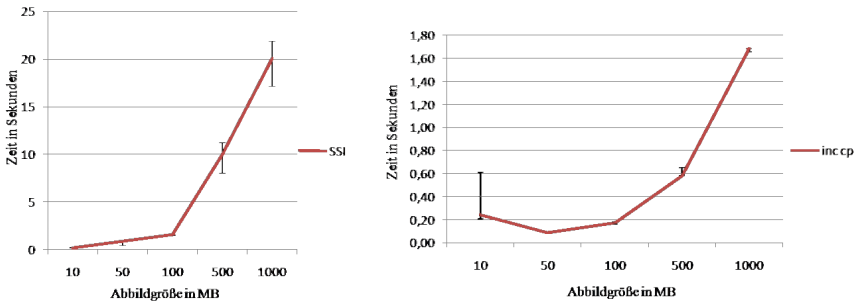

Abbildung 8.18: Reguläres vs. inkrementelles Checkpointing

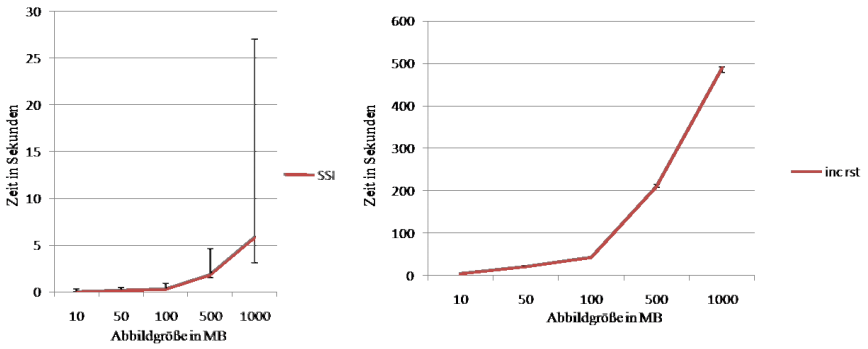

Abbildung 8.19: Regulärer versus inkrementeller Restart

des Prozessadressraums. Dies liegt darin begründet, dass pro Seite zum Checkpointing-Zeitpunkt überprüft werden muss, ob das Write-Bit gesetzt ist. In Abbildung 8.19 hingegen werden die durchschnittlichen *Wiederherstellungszeiten* beider Checkpointingstrategien dargestellt. Demnach benötigt der inkrementelle Restart mehr als das Hundertfache des regulären Restarts. Das hat folgende Gründe. Zunächst muss die Kontrollstruktur, siehe Kapitel 7.5.3, in den Kernspeicher eingelesen und dort wiederaufgebaut werden. Sie gibt Auskunft darüber, welche Speicherseite sich in welchem Abbild an welcher Position befindet. Bei diesem Szenario können die Prozessseiten über maximal zehn Abbilder verteilt sein. Daher entsteht neben dem Verwaltungsaufwand, bezüglich der Kontrollstruktur, zusätzlicher Aufwand, beispielsweise hohe Positionierungszeiten des Lesekopfes, wenn Inhalte von vielen Abbildversionen eingelesen werden müssen. Letzteres stellt ein allgemeines Problem inkrementellen Checkpointings dar.

Weiterhin muss garantiert werden, dass das Write-Bit pro Seite zurückgesetzt wurde. Ähnlich wie unkoordiniertes ist inkrementelles Checkpointing überwiegend beim Checkpointingzeitpunkt von Vorteil. Das gilt jedoch nur, wenn nur in geringem Umfang Schreibzugriffe von Seiten der Anwendung her getätigt wurden.

Um den hohen Positionierungsaufwand zu reduzieren können mithilfe einer Garbage Collection nicht mehr benötigte Abbilder entfernt und gültige Abbilder enger aneinander an-

geordnet werden.

8.6 GCA - Recovery-Line-Berechnung

Der Koordinierungsaufwand kann mit unkoordiniertem Checkpointing eliminiert werden. Die Protokollimplementierung wurde, aufbauend auf einer verteilten Anwendung mit vier Client-Job-Einheiten, die in Abständen von 3s Nachrichten an eine Server-Job-Einheit senden, evaluiert. Eine Client-Job-Einheit wurde von einem SSI-Checkpointer, die Server- und verbleibenden Client-Job-Einheiten von vier BLCR-Checkpointern gesichert und wiederhergestellt.

Bei unabhängigem Checkpointing ist der *Wiederherstellungsaufwand* ein wichtiges Leistungskriterium, weil erst beim *Restart* alle aufgezeichneten Abhängigkeiten eingelesen und verarbeitet[9] werden. Letzteres wird mithilfe der Recovery-Line-Berechnung durchgeführt. Weil der Wiederherstellungsaufwand von der Abhängigkeitsverarbeitung beeinflusst wird, wurden Messsungen mit einer unterschiedlichen Anzahl gesendeter Nachrichten, jedoch gleichbleibender Paketgröße (1 kB), vorgenommen. Tabelle 8.4 stellt die ermittelten Zeiten der Recovery-Line-Berechnung dar. Jede Job-Einheit entscheidet mithilfe einer Zufallszah-

Tabelle 8.4: Recovery-Line-Berechnung mit wachsender Nachrichten-Anzahl

Anzahl Nachrichten	100	200	300	400	500
Min	0,085s	0,094s	0,101s	0,123s	0,150s
Avg	0,091s	0,095s	0,105s	0,134s	0,176s
Max	0,095s	0,097s	0,107s	0,168s	1,347s

lenfunktion darüber, wann sie gesichert wird. In diesem Szenario ist, wie zu erwarten war, beobachtbar, dass *mit zunehmendem Nachrichtenaustausch mehr Abhängigkeiten* verarbeitet werden müssen, bevor ein globaler Checkpoint ermittelt werden kann. Die Differenz des Fehlerzeitpunkts zum Zeitpunkt des letzten global, konsistenten Zustands vergößert sich mit zunehmendem Nachrichtenaustausch. Werden Checkpoints unabhängig davon initiiert, wie Nachrichten ausgetauscht werden, entstehen viele lokale Checkpoints, die in diversen Kombinationen miteinander zu verlorenen und verwaisten Nachrichten beim Restart führen würden.

Diese Messdaten sind charakteristisch für das Kommunikations- und Checkpointmuster der in diesem Szenario verwendeten Testanwendung. Der Zusammenhang von Recovery-Line-Berechnungsaufwand und Nachrichtenaustausch muss bei einer anderen Testanwendung nicht gelten.

[9] Die Sicherungszeiten einer einzelnen Job Einheit wurden bereits unter 8.2 dargelegt.

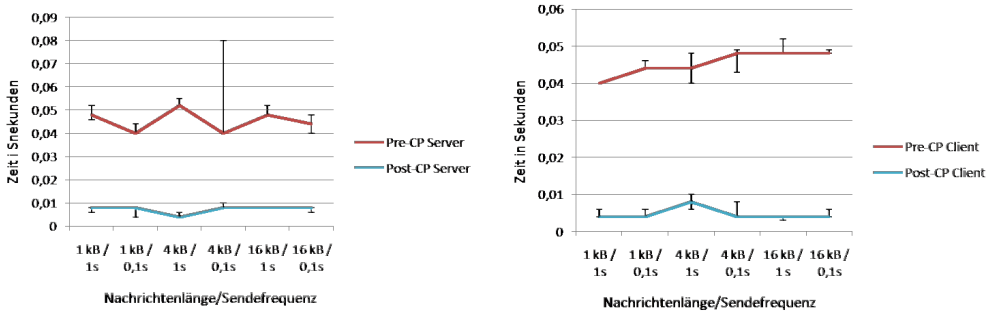

Abbildung 8.20: GKS mit explizitem Kanalab- und Wiederaufbau

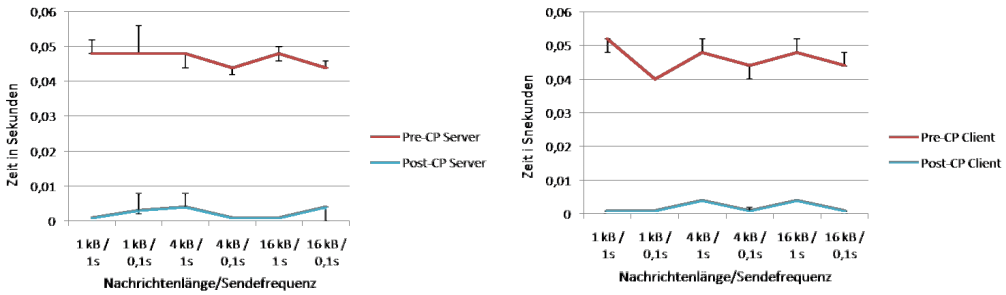

Abbildung 8.21: GKS ohne expliziten Kanalab- und Wiederaufbau

8.7 GCA - Gridkanalzustandssicherung

Die folgenden Messungen beziehen sich auf die in Kapitel 6 vorgestellte Gridkanalzustandssicherung (GKS). In einer verteilten Anwendung sendet eine Server-Job-Einheit periodisch Nachrichten an 15 Client-Job-Einheiten. Jede Job-Einheit befindet sich auf einem separaten Gridknoten. Es wurde untersucht, ob die Nachrichtenlänge entscheidenden Einfluss auf die Performanz der Kanalsicherung hat. Dies ist interessant, da hierdurch mehr oder weniger Kanalinhalte herausgeschoben und gepuffert werden müssen. Zudem wurde gemessen, wie sich die Kanalsicherung bei unterschiedlichen Sendefreqenzen verhält. Die Sicherung sollte unabhängig davon, wie viele Daten noch zu senden oder empfangen sind, *unmittelbar* vonstatten gehen.

Die Abbildung 8.20 stellt die durchschnittliche Dauer der Pre- und Post-Checkpointphase für einen Server und 15 Clients dar. In diesem Szenario werden die 15 Kanäle ab- und wiederaufgebaut. Es ist ersichtlich, dass die Pre-Checkpoint-Phase bei Client und Server mehr Zeit kostet, als die Post-Checkpoint-Phase. Dies ist nachvollziehbar, da der gesamte Kanalsicherungsaufwand in der Pre-Checkpointing-Phase stattfindet, inklusive der Kanalsicherungskoordinierung der involvierten Kontrollthreads, der Kanalleerung und des Kanalabbaus. In der Post-Checkpointing-Phase werden lediglich Kanäle wiederaufgebaut und gepufferte Nachrichten eingespeist. Die Nachrichtenlänge hat bei diesen Messungen kei-

nen Einfluss auf die Sicherungsdauer, weil sie bei 1 kB, 4 kB und 16 kB bei etwa 0,048s beträgt. Es liegt nahe, dass die Bandbreite von 1 Gbit/s die Ursache hierfür darstellt. Obwohl der Server 15 Kanäle zu bedienen hat, erkennt man, dass er fast genauso viel Zeit, wie ein Client benötigt. Dies ist auf die Nebenläufigkeit der Implementierung, wonach jeder Kanal auf Client, Server und Master-Seite von einem separaten Thread behandelt wird, zurückzuführen.

Abbildung 8.21 stellt analog zu oben die Pre- und Post-Checkpoint-Phasen von Server und Client dar, ohne, dass die Kanäle ab- und wiederaufgebaut werden. Weil keine Kanäle wiederaufgebaut werden müssen, verkürzen sich die Post-Checkpoint-Phasen der Clients und des Server im Vergleich zu den Werten aus Abbildung 8.20.

9 Zusammenfassung

9.1 Resultat

In dieser Arbeit wurde gezeigt, wie Fehlertoleranz in heterogenen Grid-Umgebungen realisiert werden kann. Hierzu wurde eine Grid-Checkpointing-Architektur (GCA) konzipiert und realisiert. Ein neuer Aspekt hierbei ist, dass heterogene Checkpointer-Pakete in die GCA integriert werden, um einfache und verteilte Jobs zu sichern und wiederherzustellen.

Bei der Analyse derzeit existierender Checkpointer-Pakete wird ersichtlich, dass die überwiegende Mehrheit nur eine Untermenge existierender Prozessressourcen wie Prozessgruppen, IPC-Objekte wie Sockets, Pipes und Nachrichtenwarteschlangen, Dateien et cetera, sichern und wiederherstellen kann. Um konsistente Abbilder zu erzeugen, obliegt es der Verantwortung der GCA, Anwendungen und Checkpointer miteinander in Bezug zu setzen.

Ausgehend vom Job-Checkpointer adressiert jeder involvierte Job-Einheit-Checkpointer einen oder mehrere zugrunde liegende Checkpointer-Pakete einheitlich und transparent mithilfe der *Uniformen Checkpointer-Schnittstelle* (UCS). Die UCS bildet das Kernelement der GCA und wird pro Checkpointer-Paket mithilfe einer Übersetzungsbibliothek implementiert. Die UCS ermöglicht, dass koordiniertes, unabhängiges und inkrementelles Checkpointing auf Basis heterogener Checkpointer realisiert werden können. Sie bildet Grid-Semantiken auf jene der Checkpointer-Pakete, beziehungsweise von Linux ab, ermittelt Abhängigkeiten zwischen Abbildern, forciert, dass heterogene Checkpointer-Pakete miteinander kooperieren und realisiert Callback-Management.

Zusätzlich zum systeminitiierten, generischen Job-Checkpointing ermöglicht die GCA selbstdefinierte und selbstgesteuerte Fehlertoleranz. Hierzu kann das Wissen eines Anwendungsprogrammierers einbezogen werden, der sicherungsrelevante Ressourcen eines Jobs kennt, sodass Checkpointing-Effizienz erhöht wird. Weiterhin können Checkpoints über eine entsprechende Schnittstelle vom Job aus initiiert werden.

Die Messungen zeigen, dass der GCA-Sicherungs- und Wiederherstellungsaufwand bei verteilten Jobs hauptsächlich vom Ein- und Ausgabeverhalten verwendeter Speichermedien und der Netzwerkinfrastruktur abhängt. Der GCA-Aufwand ist im Vergleich zu nativen Checkpointer-Paketen vernachlässigbar.

Callbacks werden im Checkpointing-Kontext eingesetzt, um anwendungsgesteuerte Optimierungen beim Checkpointing und Restart zu ermöglichen und funktionale Beschrän-

kungen von Checkpointer-Paketen aufzuheben. Die GCA stellt eine Callback-Infrastruktur zur Verfügung, um Callbacks einheitlich zu registrieren, unabhängig von einem zugrunde liegenden Checkpointer-Paket. Callbacks lassen sich jedoch auch anwendungstransparent registrieren. Dies wird eingesetzt, um gridknotenübergreifende Ressourcen wie Kommunikationskanäle *generisch* zu sichern und wiederherzustellen, ohne die Anwendung oder den Checkpointer dahingehend anpassen zu müssen.

Weil das native Betriebssystem elementare Prozesse nicht auf einen Job abbildet, jedoch Checkpointer-Pakete auf Prozessgruppen arbeiten, integriert die GCA ein eigenes Prozessgruppen-Management. Hierdurch wird abgesichert, dass ausschließlich zum Job gehörende Prozesse gesichert und wiederhergestellt werden. Die GCA verhindert, dass Ressourcenkonflikte beim Restart auftreten, indem sie Checkpointer-Pakete wie OpenVZ unterstützt, die auf leichtgewichtigen Virtualisierungsmechanismen aufbauen.

Die Gridkanalsicherung (GKS) der GCA sichert in-Transit Nachrichten von Kommunikationskanälen, welche Teil des globalen Zustands eines verteilten Jobs sind. Der Vorteil von GKS besteht darin, dass hiermit auch Single-Node-Checkpointer genutzt werden können, um verteilte Anwendungen zu sichern und wiederherzustellen. GKS unterstützt Kernel- und Bibliothekscheckpointer in Bezug auf die Sicherung und Wiederherstellung von Sockets, die unter Linux gemeinschaftlich von verwandten Prozessen genutzt werden können. GKS wird mithilfe von Callbacks und des Library-Interposition-Mechanismus' anwendungs-, betriebssystem- und checkpointertransparent integriert, im Gegensatz zu existierenden Live-Migrations-Ansätzen. Die durchgeführten Messungen belegen, dass die Gridkanalsicherung effizient ist und den Gesamtsicherungsprozess nur minimal verzögert.

Adaptives Checkpointing zielt auf eine verbesserte Fehlertoleranzeffizienz ab, indem das Checkpointingverhalten dynamisch an das System- und Anwendungsverhalten angepasst wird. Im Gegensatz zur Verwendung *eines einzigen* Checkpointingprotokolls über die gesamte Job-Lebenszeit hinweg, ermöglicht die GCA erstmals, beidseitig zwischen koordiniertem und unkoordiniertem Checkpointing zu wechseln, um auf Abhängigkeiten reagieren zu können, welche die Anwendungsberechnung und den Restart beeinflussen.

Zusätzlich unterstützt die GCA den beidseitigen Wechsel zwischen vollständigem und inkrementellem Checkpointing in Bezug auf LinuxSSI, um ein unterschiedliches Job-Schreibverhalten im Abbild reflektieren zu können. Bei der verwendeten Testanwendung ist inkrementelles Checkpointing vier Mal schneller, als bei der Sicherung des gesamten Anwendungsadressraums. Der Restart hingegen kann unter Umständen deutlich langsamer sein, abhängig davon, wieviele Abbilddateien gelesen werden müssen.

9.2 Ausblick

Auf absehbare Zeit wird es keinen Universal-Checkpointer geben, der auf allen Systemen lauffähig ist, sämtliche Checkpointingprotokolle und -strategien unterstützt, Abbild-

portabilität gewährleistet, verschiedene Container-Technologien implementiert et cetera. Daher wird eine Architektur, die heterogene Checkpointer integriert, auch zukünftig wichtig sein.

Die Kanalsicherung kann weiter optimiert werden, indem eine nebenläufige Lösung verwendet wird. Hierdurch müssen verteilte Jobs aufgrund von Kanalzustandsicherungen dann nicht mehr angehalten werden.

Da Virtualisierungslösungen in vielen Bereichen eingesetzt werden, ist es sinnvoll, die GCA zu erweitern, um auch *Virtuelle Maschinen* und weitere Container-basierte Checkpointer zu integrieren.

Neben typischen Anwendungen des High Performance Computing (HPC)-Bereichs kann die GCA auch in Richtung interaktiver zwei- und dreidimensionale Graphikanwendungen ausgebaut werden.

Auch weitere gemeinsam nutzbare Ressourcen, wie Dateien und Segmente, können durch zusätzliche Sicherungsmechanismen angegangen werden, die, ähnlich der GKS, auf Checkpointer-Kooperationen basieren.

Während der Ursprung des Grid-Computings im wissenschaftlichen Hochleistungsrechnen liegt und darauf fokussiert, dass Ressourcen verfügbar sind, werden beim sogenannten Cloud-Computing Dienste angeboten, die im Sinne der 'pay-as-you-go'-Semantik flexibel genutzt werden können. Gerade weil im Gegensatz zum Grid-Computing verstärkt ökonomische Interessen vorliegen, müssen wichtige Anwendungszustände von Cloud-Computing-Kunden vor deren Verlust geschützt werden, wenn Hard- und/oder Softwarekomponenten ausfallen.
Anders als *Private Clouds* fördern *Public Clouds*, dass IT-Infrastrukturen aus einem Unternehmen ausgelagert werden können (engl. outsourcing), um Kosten zu sparen. Hierbei werden Rechen-, Speicher- und Daten-Ressourcen, unter Einsatz verschiedener Formen von Virtualisierung und unter Berücksichtigung sicherheitsrelevanter Aspekte, in Datenzentren bereitgestellt (IaaS = infrastructure as a service). Auch in diesem Umfeld kann die GCA innerhalb virtualisierter Instanzen verwendet werden oder als ein IaaS-Dienst.

10 Summary

In this phd thesis a grid checkpointing architecture (GCA) has been designed and realised to provide fault tolerance in a heterogeneous grid environment. The unique approach of this architecture is to integrate existing heterogeneous checkpointer packages in order to checkpoint and restore single and distributed jobs.

The analysis of existing checkpointer packages showed a great diversity of their ability to consistently save and restore numerous process resources such as process groups, inter process communication objects, files, etc. Thus, creating consistent checkpoint images requires the GCA to identify checkpointers which are compatible with resources of a given job.

The top-most GCA component, the job checkpointer, addresses each involved job-unit checkpointer which in turn addresses an underlying checkpointer-package in a uniform and transparent fashion via a so-called Uniform Checkpointer Interface (UCS). The UCS is the core element of the GCA and is implemented in a checkpointer-bound translation library. THe UCS implements the coordinated, independent and incremental checkpointing protocol that can be executed on top of heterogeneous checkpointer-packages. It maps semantics from the grid to the checkpointer-package and Linux level, it detects dependencies between checkpointing images, has heterogeneous checkpointer-packages cooperate with each other and realises callback management.

The GCA provides job-defined and job-controlled fault tolerance, additionally to system-initiated, generic job checkpointing. Thus, an application developer can integrate its knowledge about checkpointing relevant resources to reduce checkpointing overhead and thus increase checkpointing efficiency. A GCA interface allows jobs to independently initiate checkpoints.

The GCA measurement evaluation showed, that the checkpoint/restart overhead primarily results from the input and output behaviour of underlying network infrastructure and storage components. Furthermore, the GCA overhead is minimal compared to native checkpointer packages.

Callbacks are used in checkpointing to optimize job-controlled fault tolerance and to overcome checkpointer-bound restrictions. The GCA provides a callback infrastructure for jobs to uniformly register callbacks, independent of an underlying checkpointer-package. Furthermore, callbacks can be registered in an application transparent fashion. The latter is used to genericallly save and restore resources, spread across multiple nodes, e.g. communication channels.

Since a job abstraction is unknown by the underyling native operating system, the GCA integrates a grid-specific process group management to map native processes onto a job. Furthermore, the GCA must apply checkpointer specific operations at job submissione.g. creation of a process group, such that a checkpointer can reference all job processes at checkpoint time. The latter is due checkpointers being applicable only onto process gropups instead of a given process list. The GCA integrates container-based checkpointer-packages, such as OpenVZ. They are based on lightweigth virtualisation which can be used to eliminate potential resource conflicts at restart.

The GCA inherent grid channel checkpointing (GCC) saves in-transit messages of communication channels that constitute a globally constistent state of a distributed job. One of the major GCC benefits is the integration of *single node* checkpointer-packages that can be used to checkpoint and restart *distributed jobs*. GCC supports kernel and library checkpointer-packages regarding checkpointing and restarting sockets shared among hierarchical processes under Linux. GCC is integrated into a job, operation system and checkpointer in a transparent fashion based on callbacks and the library interposition mechanism. Measurements prove the GCC to be efficient. It does not delay the overall checkpointing process in a meaningful way.

Adaptive checkpointing targets to improve fault tolerance efficiency. Therefore, it adapts the checkpointing behaviour dynamically to the system and job behaviour. Opposite to the mere usage of just one checkpointing protocol across the job life time, GCA enables to mutually switch between coordinated and independent checkpointing to react on the amount of existing dependencies.

Furthermore, it supports switching between incremental and full checkpointing reagrding LinuxSSI. Thus, it can react on a varying job write behaviour.. Testing incremental checkpointing turned out to be four times faster then checkpointing the whole address space. However, the restart tends to be significantly slowlier, dependent on the number of checkpoint image to be read.

It is unlikely for a universal checkpointer to come up in the near future, that can be executed on every operating system, that supports all checkpointing protocols, that provides image portability, implements heterogeneous container technologies, etc. That's why a grid checkpointing architecture, integrating heterogeneous checkpointer packages, will be of great interest in the future.

Grid channel checkpointing (GCC) can still be optimized. If there is a concurrent solution distributed jobs do not need to be synchrionized and thus paused for channel content saving.

Since virtualisation solutions are used in many areas it is of great benefit to extend the GCA by virtual machine and container-based checkpointe packages.

Besides typical high performance computing (HPC) applications the GCA can be extended to support interactive two and threedimensional graphical applications.

Furthermore, there are additional ressources, that can be used by multiple, distributed processes at one time, e.g. files and segments, which can be supported by a mechanism that requires checkpointer cooperation, similar to GCC.

While grid computing, that is based on scientific high performance computing, focusses on resource availability, cloud computing provides services that can be used in a 'pay-as-you-go' manner. Since cloud computing emphasizes the economical usage of resources, significant states of customer applications must be prevented from losses in case software and/or hardware components fail. Opposite to private clouds do public clouds encourage enterprises to outsource IT infrastructures for financial purposes. Computing, storage and data resources are provided by data centers based on various forms of virtualisation and with regards to security. This is called infrastructure as a service (IaaS). The GCA can be used within virtualised instance or even as a IaaS service itself.

A Messwerttabellen

A.1 Checkpointing mit nativem BLCR, MTCP und SSI

A.1.1 Checkpointing

	10 MB	50 MB	100 MB	500 MB	1000 MB
BLCR Min	0,244016238	1,220081263	2,424157729	12,912839429	26,145720163
BLCR Avg	0,258417203	1,281685366	2,580168071	13,361617561	29,608347588
BLCR Max	0,276018366	1,372091388	2 ,656173016	13,900890608	31,622079646
MTCP Min	0,380203413	0,904657011	1,644432656	6,956193201	28,963932946
MTCP Avg	0,380859554	0,933298480	1,736513540	7,214888086	29,631443
MTCP Max	0,384230179	0,9488314595	1,892550169	7,374357870	30,841605468
SSI Min	0,031134243	0,724739435	1,519700378	9,992831409	20,090341117
SSI Avg	0,153114985	0,826717471	1,704636366	10,913834903	29,08294456
SSI Max	0,155787406	1,204081076	2,431680513	13,379009584	29,287621054

Tabelle A.1: Checkpointing mit nativem BLCR, MTCP und LinuxSSI (in Sekunden)

A.1.2 Restart

	10 MB	50 MB	100 MB	500 MB	1000 MB
BLCR Min	0,136009060	0,660043962	1,308085302	6,736443317	41,226710629
BLCR Avg	0,141609378	0,743249503	1,392890832	6,991656088	41,430723724
BLCR Max	0,152010125	0,808053820	1,576102779	7,252471491	41,614735210
MTCP Min	0,148607770	0,388334532	0,688443112	3,052679934	3,488239263
MTCP Avg	0,149123562	0,429189376	0,7684317632	3,352000831	9,612175332
MTCP Max	0,1528209286	0,512344291	0,960580499	3,965182027	35,197048552
SSI Min	0,033608555	0,148180943233	0,2971512444	1,518551621	3,091803387
SSI Avg	0,033816115	0,184422166	0,361682481	1,824884431	5,811433133
SSI Max	0,0353138839	0,46130317152	0,915850189	4,603944967	27,0312465127

Tabelle A.2: Restart mit nativem BLCR, MTCP und LinuxSSI (in Sekunden)

A.2 GCA-Checkpointing einer Job Einheit

A.2.1 Checkpointing

	10 MB	50 MB	100 MB	500 MB	1000 MB
BLCR Min	0,440173287	1,444545475	2,713083529	13,056585541	27,072334651
BLCR Avg	0,563217465	1,504546659	2,755497067	13,277579794	27,174432584
BLCR Max	0,684264531	1,552556709	2,801113370	13,416881979	27,130256192
MTCP Min	0,532159554	1,072298480	1,736513540	7,046193201	28,963932946
MTCP Avg	0,604179917	1,115501286	1,828534608	7,247008996	31,742787679
MTCP Max	0,804230179	1,148314595	1,892550169	7,374357870	34,484605468
SSI Min	0,311114985	0,914174711	1,704636366	10,913834903	2,908294456
SSI Avg	0,355571390	0,928439916	1,925945473	11,815493575	20,914880881
SSI Max	0,417874064	0,964081076	2,431680513	13,379009584	29,287621054

Tabelle A.3: GCA-Checkpointing einer Job-Einheit (in Sekunden)

A.2.2 Restart

	10 MB	50 MB	100 MB	500 MB	1000 MB
BLCR Min	0,2182128292	0,216286346	1,468480797	6,954888366	39,613103959
BLCR Avg	0,3729605385	0,80842808442	1,554196941	8,997111946	40,044892026
BLCR Max	0,80994943	1,1424757404	1,612627669	16,754255577	40,606547845
MTCP Min	0,588123562	0,852189376	1,204317632	3,713000831	0,876239263
MTCP Avg	0,636961153	1,1782344291	1,562814381	3,937060981	15,708613534
MTCP Max	0,728209286	1,512677632	2,160580499	4,405182027	50,930485520
SSI Min	0,181663115	0,284422166	0,446824813	1,518551621	4,181803387
SSI Avg	0,523963181	0,600787727	1,280939050	4,628854486	5,480286919
SSI Max	1,713138839	1,830317152	3,688850189	16,846944967	10,672465127

Tabelle A.4: GCA-Restart einer Job-Einheit (in Sekunden)

A.3 GCA Checkpointing - mehrere Job Einheiten

Konstellation SSI-MAJ entspricht sieben SSI Clustern (jeweils zwei physikalische Knoten), einem Knoten mit BLCR und einem mit MTCP.
Konstellation SSI-MIN entspricht einem SSI Cluster, sieben Knoten mit BLCR und sieben mit MTCP.

A.3.1 Checkpointing

	90 MB	450 MB	900 MB	4500 MB	9000 MB
Min	0,2409010025	1,0854808178	2,1105848950	12,5150707895	28,1894438028
Avg	0,3256296503	1,1711418651	2,2551878042	12,9399041325	28,9078365814
Max	0,3956421807	1,2741613613	2,5180401680	13,2492506898	29,6262293599

Tabelle A.5: GCA-Checkpointing von SSI-MAJ (in Sekunden)

	150 MB	750 MB	1500 MB	7500 MB	15000 MB
Min	0,2409010025	2,4295965910	6,2732223939	31,8691639883	49,1639100004
Avg	0,5737212195	2,5051122632	6,3492135065	32,4960672554	60,4533551281
Max	0,6302062897	2,5806279353	6,4313287166	33,1229705226	69,6930081677

Tabelle A.6: GCA-Checkpointing mehrerer Job-Einheiten (SSI-MIN) (in Sekunden)

A.3.2 Restart

	90 MB	450 MB	900 MB	4500 MB	9000 MB
Min	0,746354332	2,0003199354	1,973989498	83,05665403	578,743336066
Avg	1,342663712	2,4956706956	16,405565916	111,662803041	635,781547098
Max	1,936932050	2,6802190491	58,647350939	310,975409582	692,819758130

Tabelle A.7: GCA-Restart mehrerer Job-Einheiten (SSI-MAJ) (in Sekunden)

	150 MB	750 MB	1500 MB	7500 MB	15000 MB
Min	2,432736275	7,506345380	20,8801609592	110,0488725627	2350,993465888
Avg	2,584560695	8,779308367	21,0674994046	110,6441202635	2409,518537696
Max	2,736385115	10,052271354	21,2548378499	111,2393679644	2412,222397186

Tabelle A.8: GCA-Restart mehrerer Job-Einheiten (SSI-MIN) (in Sekunden)

A.4 Ein-/Ausgabe unter NFS mit variierenden Blockgrößen

Die Testanwendung besitzt eine Adressraumgröße von 100 MB und wurde mit BLCR gesichert, beziehungsweise wiederhergestellt.

A.4.1 Checkpointing

	32768 Bytes	103.000 Bytes	1048576 Bytes
Min	56,113384648	62,732223939	58,239430292
Avg	64,125131996	63,492135065	64,485530816
Max	73,111509446	64,313287166	67,488468573

Tabelle A.9: Checkpointing auf NFS (in Sekunden)

A.4.2 Restart

	32768 Bytes	103.000 Bytes	1048576 Bytes
Min	249,433364188	208,801609592	211,984352749
Avg	253,241182739	210,674994045	217,595305594
Max	259,951372535	212,548378499	226,434255394

Tabelle A.10: Restart von NFS (in Sekunden)

A.5 Ein-/Ausgabe auf der lokalen Festplatte

A.5.1 Checkpointing

	10 MB	50 MB	100 MB	500 MB	1000 MB
Min	2,488182393	9,492687681	18,293307858	88,682369598	176,508516624
Avg	2,610861390	9,510023665	18,311985845	88,821020061	176,791218287
Max	2,852218845	9,520691269	18,325325736	89,050319957	177,288617539

Tabelle A.11: Checkpointing auf lokale Festplatte (in Sekunden)

A.5.2 Restart

	10 MB	50 MB	100 MB	500 MB	1000 MB
Min	2,788199707	12,804916829	24,557751016	125,732953804	251,137871733
Avg	3,078887197	12,890256278	25,191131826	126,317007498	251,584545844
Max	3,260233550	12,996930474	25,545828624	126,705040065	252,461894446

Tabelle A.12: Restart von lokaler Festplatte (in Sekunden)

A.6 Inkrementelles Checkpointing

A.6.1 Checkpoint

	10 MB	50 MB	100 MB	500 MB	1000 MB
Min	0,970276566	0,889223334	0,941013027	1,436719559	2,503039425
Avg	1,150407230	0,962227026	0,953909633	1,651763974	2,630039251
Max	1,803933913	1,017902298	1,114042911	2,001229618	2,946503000

Tabelle A.13: Inkrementeller Checkpoint (in Sekunden)

A.6.2 Restart

	10 MB	50 MB	100 MB	500 MB	1000 MB
Min	0,970276566	0,889223334	0,941013027	1,436719559	2,503039425
Avg	1,150407230	0,962227026	0,953909633	1,651763974	2,630039251
Max	1,803933913	1,017902298	1,114042911	2,001229618	2,946503000

Tabelle A.14: Inkrementeller Restart (in Sekunden)

B Grid-Checkpointing Metadaten

B.1 JSDL-Erweiterung

```
<?xml version="1.0" encoding="UTF-8"?>
  <JobDefinition xmlns="http://schemas.ggf.org/jsdl/2005/10/jsdl">
    <JobDescription>
      <JobCheckpointing>
        <Initiator>System</Initiator>
        <ProtocolManagement>
                <Name>CoordinatedCheckpointing</Name>
                        <Parameter>Incremental</Parameter>
        </ProtocolManagement>
        <CheckpointFileManagement>
                <ReplicationLevel>5</ReplicationLevel>
                <Compression>LZW</Compression>
        </CheckpointFileManagement>
        <JobCheckpointerMatching>
                <MultiThread>Yes</MultiThread>
                <MultiProcess>Yes</MultiProcess>
                <IPCType>POSIX</IPCType>
                <IPC-shm>Yes</IP-shm>
                <Sockets>Yes</Sockets>
                <Pipe>Yes</Pipe>
        </JobCheckpointerMatching>
        <Checkpointer>BLCR-v0.8.2-64</Checkpointer>
        <Container>
                <Type>Cgroups</>Type>
                <Subsystem>Name</Subsystem>
        </Container>
      </JobCheckpointing>
    </JobDescription>
  </JobDefinition>
```

Listing B.1: Checkpointing-Metadaten für erweitertes JSDL-Format

C Lebenslauf

Name:	Mehnert-Spahn
Vorname:	John Hans-Jürgen
Geburtsdatum:	20. März 1979
Nationalität:	Deutsch
Familienstand:	ledig
Adresse:	Altenbrückstr. 43A 40599 Düsseldorf
Büro:	Heinrich-Heine Universität Düsseldorf Geb. 25.12.01 Email: John.Mehnert-Spahn@uni-duesseldorf.de

Ausbildung:

1985-1991	Polytechnische Oberschule, Seebach
1991-1993	Ernst-Abbe Gymnasium, Eisenach
1993-1998	Musikgymnasium Schloss Belvedere, Weimar
1998-1999	Zivildienst, Weimar
1999-2000	Auslandsjahr
Okt. 2000 - Juni 2006	Studium der Angewandten Informatik an der Universität Siegen
Juli 2004 - Januar 2005	Auslandsemester am Royal Melbourne Institute of Technology, Melbourne (Australien)
August 2005-März 2006	Diplomarbeit bei der DaimlerChrysler AG, Ulm
Juni 2006	Diplom
seit Oktober 2006	wissenschaftlicher Mitarbeiter in der Abteilung für Betriebssysteme an der Heinrich Heine-Universität, Düsseldorf

Praktische Tätigkeiten:

July 2001-August 2001	Praktikant bei der Aventis Behring GmbH, Liederbach
Januar 2003 - Mai 2004	studentische Hilfskraft am Fraunhofer Institut für Medienkommunikation (IMK) Schloss Birlinghoven, Sankt Augustin
April 2006	studentische Hilfskraft bei DaimlerChrysler AG Forschung und Technik, Ulm

Veröffentlichungen:

Checkpointing and Migration of Communication Channels in Heterogeneous Grid Environments
J. Mehnert-Spahn, M. Schöttner
The 10th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2010)
Busan, Südkorea
21.-23. Mai 2010

The Architecture of the XtreemOS Grid Checkpointing Service
J. Mehnert-Spahn, T. Ropars, M. Schöttner, C. Morin
Euro-Par 2009
Delft, Niederlande
25.-28. August 2009

Incremental Checkpointing for Grids
J. Mehnert-Spahn, E. Feller, M. Schöttner
Linux Symposium
Montreal, Kanada
12.-17. Juli 2009

Checkpointing Process Groups in a Grid Environment
J. Mehnert-Spahn, M. Schöttner, C. Morin
Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2008)
Dunedin, Neuseeland
1.-4. Dezember 2008

Anwendungsübergreifende Benutzerinteraktion in virtuellen Umgebungen
J. Mehnert-Spahn, S. Steck, A. Kolb
9. IFF-Wissenschaftstage
Magdeburg, Deutschland
Juni 2006

An Open and Extensible Framework for Visualization
C. Bastuck, T. Hambürger, T. Hof, M. Keller, P. Kohlmann, J. Mehnert-Spahn, S. Nowak
Fachwissenschaftlicher Informatik-Kongress
Schloss Birlinghoven, St. Augustin bei Bonn, Deutschland
8.-9. April 2005

Weitere Tätigkeiten:

Poster-Session bei CCGrid 2008
19.-22. Mai 2008
Lyon, Frankreich

Invited Speaker PDCAT08
Dunedin, Neuseeland
1.-4. Dezember 2008

XtreemOS Summer School 2009
University of Oxford
Oxford, England
7.-11. September 2009

Literaturverzeichnis

[1] Project: Btrfs, 2007.

[2] Bouteiller A., T. Ropars, Bosilica G., C. Morin, and Dongarra J. Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure recovery. 2009.

[3] S. Agarwal. Distributed checkpointing of virtual machines in xen framework. 2008.

[4] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM.

[5] Adnan Agbaria and Roy Friedman. Virtual-machine-based heterogeneous checkpointing. *Softw. Pract. Exper.*, 32(12):1175–1192, 2002.

[6] Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, July 2003.

[7] M. Aldinucci, M. Danelutto, G. Antoniu, and M. Jan. Fault-tolerant data sharing for high-level grid: a hierarchical storage architecture. In *Achievements in European Research on Grid Systems*, pages 67–8. Springer Verlag, 2008.

[8] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Comput.*, 28(5):749–771, 2002.

[9] W. Almesberger. Another form of tcpcp, 2006. http://tcpcp2.sourceforge.net/. Zuletzt besucht am 13.05.2010.

[10] Lorenzo Alvisi. *Understanding the message logging paradigm for masking process crashes*. PhD thesis, Ithaca, NY, USA, 1996.

[11] Lorenzo Alvisi and Keith Marzullo. Trade-offs in implementing causal message logging protocols. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 58–67, New York, NY, USA, 1996. ACM.

[12] Lorenzo Alvisi and Keith Marzullo. Trade-offs in implementing causal message log-

ging protocols. In *In Proceedings of the 1996 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing Systems (PODC*, pages 58–67, 1996.

[13] Lorenzo Alvisi, Sriram Rao, Syed Amir Husain, Meland Asanka de, and Elmootazbellah Elnozahy. An analysis of communication-induced checkpointing. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 242, Washington, DC, USA, 1999. IEEE Computer Society.

[14] N. Stone amd D. Simmel and T. Kielmann. An architecture for grid checkpoint and recovery (gridcpr) services and a gridcpr application programming interface. Technical report, 2005.

[15] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[16] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[17] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. Juxmem: An adaptive supportive platform for data sharing on the grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, September 2005.

[18] R. Badia, R. Hood, T. Kielmann, C. Morin, S. Prickles, M. Sgaravatto, P. Stodghill, N. Stone, and H. Y. Yeom. Use-cases for grid checkpoint and recovery. Technical report, 2005.

[19] Sujata Banerjee, Sujoy Basu, Shishir Garg, Sukesh Garg, Sung-Ju Lee, Pramila Mullan, and Puneet Sharma. Scalable grid service discovery based on uddi. In *MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing*, pages 1–6, New York, NY, USA, 2005. ACM.

[20] S. Bhargava and S.-R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed system - an optimistic approach. In *proceedings of the Symposium onReliable Distributed Systems*, pages 3–12, 1988.

[21] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42(5):104–113, 2008.

[22] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM.

[23] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Capello. Coordinated checkpoint versus message log for fault tolerant mpi. In *in IEEE International Conference on Cluster Computing (Cluster 2003). IEEE CS*, pages 242–250. Press, 2003.

[24] Aurelien Bouteiller, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. Mpich-v project: A multiprotocol automatic fault-tolerant mpi. *IJHPCA*, 20(3):319–333, 2006.

[25] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, November 2006.

[26] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algrithm. In *Proceeding of the IEEE International Symposium of Reliabiltiy, Distributed Software and Databases*, pages 207–215, December 1984.

[27] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. *SIGARCH Comput. Archit. News*, 32(5):235–247, 2004.

[28] K. Byoung-Jip. Comparison of the existing checkpoint systems. October 2005. https://lists.linux-foundation.org/pipermail/clusters_sig/attachments/20051025/88ed8b30/Comparison-CR-0001.pdf. Zuletzt besucht am 13.05.2010.

[29] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[30] Z. Chen. Adaptive checkpointing. *Journal of Communications*, Vol 5(No 1):81–87, 2010.

[31] R. Chinnici, J.-J. Moreau, A. Rayman, and S. Weerawarana. *Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language*. W3C, 2007.

[32] Augusto Ciuffoletti, Antonio Congiusta, Gracjan Jankowski, Michal Jankowski, Ondrej Krajicek, and Norbert Meyer. Grid infrastructure architecture: A modular approach from coregrid. In *3rd International Conference on Web Information Systems and Technologies (WEBIST)*, page 9, Barcelona (Spain), March 2007.

[33] L. Clement, A. Hately, C. von Riegen, and T. Rogers. Uddi spec technical committee draft. Technical report, 2004.

[34] J. Cooperstein. Linux multithreading advances. July 2002.

[35] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 127, New York, NY, USA, 2006. ACM.

[36] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[37] Z. Di. Eliminating domino effect in backward error recovery in distributed systems. pages 243–248, 1987.

[38] Jörg Domaschka, Christian Spann, and Franz J. Hauck. Virtual nodes: a reconfigurable replication framework for highly-available grid services. In *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 107–109, New York, NY, USA, 2008. ACM.

[39] N. Dorofeev. Maxtor 6y080l0 drive review. http://ixbtlabs.com/articles2/maxtor-6y080l0/. Zuletzt besucht am 13.05.2010.

[40] Ulrich Drepper. What every programmer should know about memory. 2007. http://people.redhat.com/drepper/cpumemory.pdf. Zuletzt besucht am 13.05.2010.

[41] Ulrich Drepper. The cost of virtualization. *Queue*, 6(1):28–35, 2008.

[42] A.A. Duarte. *Radic: a powerful fault-tolerant architecture*. PhD thesis, May 2007.

[43] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. 2003.

[44] K. Echtle. *Fehlertoleranzverfahren*. Springer-Verlag, 1990.

[45] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[46] E.N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Proceedings of the Twenty Fourth International Symposium on Fault-Tolerant Computing*, pages 298–307, Juni 1984.

[47] G. Ercim. Coregrid. 2004. Zuletzt besucht am 13.05.2010.

[48] E. Feller. *Masterarbeit: Unabhängiges Checkpointing in einer heterogenen Grid-Umgebung*. 2009.

[49] E. Feller, C. Morin, and J. Mehnert-Spahn. Prototype of the first advanced version of linuxssi d2.2.10. 2009.

[50] M. Feller, I. Foster, and S. Martin. Gt4 gram: A functionality and performance study. 2007.

[51] Matthieu Fertre and Christine Morin. Extending a cluster ssi os for transparently checkpointing message-passing parallel application. In *ISPAN*, pages 364–369, 2005.

[52] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput.*, 52(2):99–112, 2003.

[53] Message Passing Interface Forum. Mpi: A message-passing interface standard version 2.1, 2008.

[54] Ian Foster. The physiology of the grid: An open grid services architecture for distributed systems integration. 2002.

[55] Ian Foster and Carl Kesselman. The globus toolkit. pages 259–278, 1999.

[56] Ian T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 1–4, London, UK, 2001. Springer-Verlag.

[57] J. Frost. Unix signals and process groups.

[58] Pascal Gallard and Christine Morin. Dynamic streams for efficient communications between migrating processors in a cluster. In *Euro-Par*, pages 930–937, 2003.

[59] F. Garcia and J. Fernandez. Posix thread libraries, February 2000. http://www.linuxjournal.com/article/3184 . Zuletzt besucht am 13.05.2010.

[60] W. Gentzsch. Sun grid engine: towards creating a compute power grid. pages 35–36, 2001.

[61] Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991.

[62] S. Gerhold, P. Schmidt, A. Weggerle, and P. Schulthess. Improved checkpoint/restart using solid state disk drives. 2009.

[63] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9, Washington, DC, USA, 2005. IEEE Computer Society.

[64] F. Gomes and A. F. Bosco. *Optimizing incremental state-saving and restoration*. PhD thesis, University of Calgary, Calgary, Alta., Canada, Canada, 1996. A.-U. Brian.

[65] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. *Int. J. Parallel Program.*, 31(4):285–303, 2003.

[66] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[67] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In *In Proc Conference on Reliable Software Technologies (invited paper*, pages 38–57. Springer Verlag, 1996.

[68] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *In Proceedings of SciDAC 2006*, June 2006.

[69] M. Helsey. Process event connector. 2005. Zuletzt besucht am 13.05.2010.

[70] M. Helsey. Container freezer v6: Reuse suspend freezer, August 2008. http://lwn.net/Articles/293642/. Zuletzt besucht am 13.05.2010.

[71] M. Helsey. Freezer-subsystem documentation. 2010. http://www.mjmwired.net/kernel/Documentation/cgroups/freezer-subsystem.txt. Zuletzt besucht am 13.05.2010.

[72] Matt Helsey. Lxs: Linux container tools, Febuary 2009.

[73] Junyoung Heo, Sangho Yi, Yookun Cho, Jiman Hong, and Sung Y. Shin. Space-efficient page-level incremental checkpointing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1558–1562, New York, NY, USA, 2005. ACM.

[74] Sternberg J. Hinze M. A-revolve: an adaptive memory-reduced procedure for calculating adjoints; with an application to computing adjoints of the instationary navier–stokes system. *Optimization Methods and Software 20(6)*, 2005. http://www.informaworld.com/10.1080/10556780410001684158.

[75] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[76] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. Xtreemfs: a case for object-based storage in grid data management. In *Proceedings of 33th International Conference on Very Large Data Bases (VLDB) Workshops*, 2007.

[77] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The xtreemfs architecture—a case for object-based file systems in grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2049–2060, 2008.

[78] IBM. mutex - bad in signal context. 2006. Zuletzt besucht am 13.05.2010.

[79] B. V. Jacobson, B. Braden, and D. Borman. Rfc 1323 tcp extensions for high performance, 1992. http://www.ietf.org/rfc/rfc1323.txt. Zuletzt besucht am 13.05.2010.

[80] G. Jankowski, R. Januszewski, R. Mikolajczak, and J. Kovacs. The grid checkpointing architecture. May 2008.

[81] G. Jankowski, R. Januszewski, R. Mikolajczak, M. Stroinski, J. Kovacs, and A. Kertesz. Grid checkpointing architecture - integration of low-level checkpointing capabilites with grid. Technical Report TR-0036, CoreGRID, May 22, 2007.

[82] Gracjan Jankowski, Radoslaw Januszewski, Rafal Mikolajczak, and Jozsef Kovacs. Scalable multilevel checkpointing for distributed applications - on the possibility of integrating total checkpoint and altixc/r. Technical Report TR-0035, Institute on Grid Information and Monitoring Services, CoreGRID - Network of Excellence, May 2006.

[83] Rüdiger Kapitza and Franz J. Hauck. Edas: providing an environment for decentralized adaptive services. In *DSM '05: Proceedings of the 2nd international doctoral symposium on Middleware*, pages 1–5, New York, NY, USA, 2005. ACM.

[84] O. Laadan. Linux checkpoint code, 2009. http://git.ncl.cs.columbia.edu/?p=linux-cr.git;a=tree;f=checkpoint. Zuletzt besucht am 02.01.2010.

[85] O. Laadan, D. Phung, and J. Nieh. Transparent checkpoint-restart of distributed applications on commodity clusters. In *IEEE International Conference on Cluster Computing (CLUSTER 2005)*, September 2005.

[86] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[87] Argonne National Laboratory. Mpich2, 2009.

[88] A. Lakhani, P. Robinson, G. Pipan, E. Yang, and I. Johnson. First specification of security services d3.5.3, 2007.

[89] F. O. Leite. Load-balancing ha clusters with no single point of failure. In *Proceedings of the 9th International Linux System Technology Conference*, pages 122–131, September 2002. Zuletzt besucht am 13.05.2010.

[90] Pierre Lemarinier, Aurelien Bouteiller, Geraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. *Int. J. High Perform. Comput. Netw.*, 2(2-4):146–155, 2004.

[91] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 79–88, New York, NY, USA, 1990. ACM.

[92] Y.-B. Lin, B.R. Preiss, and E.D. Lazowska. Dynamic checkpoint interval selection in time warp simulation.

[93] John Linn. Trust models and management in public-key infrastructures, 2000.

[94] R. Lottiaux. General kerrighed architecture, September 2007.

[95] André Luckow and Bettina Schnor. Adaptive checkpoint replication for supporting the fault tolerance of applications in the grid. In *7th IEEE International Symposium on Network Computing and Applications*, Boston, USA, 2008.

[96] C. Ma, Z. Huo, J. Cai, and D. Meng. Dcr: A fully transparent checkpoint/restart framework for distributed systems. 2009.

[97] D. Magenheimer. Transcendant memory and linux. In *Proceedings of the Linux Symposium*, July 2009.

[98] David E. Culler Matthew L. Massie, Brent N. Chun. The ganglia distributed monitoring system: design, implementation, and experience. 2003.

[99] B. Matthews, T. Cortes, Y. Yégou, T. Kielmann, D. Laforenza, C. Morin, L.P. Prieto, and A. Reinefeld. Xtreemos: a vision for a grid operating system. Technical report, 2008.

[100] J. Mehnert-Spahn and M. Schoettner. Design and implementation of basic checkpoint/restart mechanisms in linuxssi d2.2.3. 2007.

[101] John Mehnert-Spahn, Michael Schöttner, and Christine Morin. Checkpointing process groups in a grid environment. In *PDCAT '08: Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 243–251, Washington, DC, USA, 2008. IEEE Computer Society.

[102] Paul Menage. Cgroups. World Wide Web electronic publication, 2006.

[103] Rosa Anna Micillo, Salvatore Venticinque, Rocco Aversa, and Beniamino Di Martino. A grid service for resource-to-agent allocation. *Internet and Web Applications and Services, International Conference on*, 0:443–448, 2009.

[104] Jayadev Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 4:37–43, January 1982.

[105] C. Morin, A. Quin, H. Yu, S. Chitre, and J. Mehnert-Spahn. Prototype of the first advanced version of linux-xos d2.1.7, 2009.

[106] Christine Morin, Alain Gefflaut, Michel Banâtre, and Anne-Marie Kermarrec. Coma: an opportunity for building fault-tolerant scalable shared memory multiprocessors. In *ISCA '96: Proceedings of the 23rd annual international symposium on Computer architecture*, pages 56–65, New York, NY, USA, 1996. ACM.

[107] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, and Louis Rilling. Kerrighed: A single system image cluster operating system for high performance computing. In *Euro-Par*, pages 1291–1294, 2003.

[108] Abbe Mowshowitz. Virtual organization. *Commun. ACM*, 40(9):30–37, 1997.

[109] Marc-Florian Mueller, Kim-Thomas Moeller, Michael Sonnenfro, and Michael Schoettner. transactional data sharing in grids". November 2008.

[110] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for hpc with xen virtualization. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 23–32, New York, NY, USA, 2007. ACM.

[111] L. Nassif, J. M. Nogueira, M. Ahmed, R. Impey, and A. Karmouch. Agent-based negotiation for resource allocation in grid. 2005.

[112] Babar Nazir, Kalim Qureshi, and Paul Manuel. Adaptive checkpointing strategy to tolerate faults in economy based grid. *J. Supercomput.*, 50(1):1–18, 2009.

[113] Lei Ni and Aaron Harwood. An adaptive checkpointing scheme for peer-to-peer based volunteer computing work flows. In *PDCAT '08: Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 227–234, Washington, DC, USA, 2008. IEEE Computer Society.

[114] Lei Ni and Aaron Harwood. An adaptive checkpointing scheme for peer-to-peer based volunteer computing work flows. In *PDCAT '08: Proceedings of the 2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 227–234, Washington, DC, USA, 2008. IEEE Computer Society.

[115] Examples of JSDL 1.0 Documents for Parallel Jobs. I. rodero and f. guim and j. corbalan and j. labarta. 2006.

[116] University of Knoxville. Mpi: A message-passing interface standard version 2.2, September 2009.

[117] University of Zagreb. Srce: Job management system. 2007.

[118] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann Publishers, Inc., 2 edition, 1998.

[119] M. Polte, J. Simsa, W. Tantisiriroj, G. Gibson, S. Dayal, M. Chainani, and D. Kumar Uppugandla. Fast log-based concurrent writing of checkpoints. In *Proceedings of the 3rd Petascale Data Storage Workshop held in conjunction with Supercomputing '08*, Austin, TX, USA, November 2008.

[120] S. Probst. Skalierbare fehlererkennung in gridumgebungen. Master's thesis, 2008.

[121] Frank Siebenlist Rajkumar Kettimuthu, Liu Wantao and Ian Foster. Communicating security assertions over the gridftp control channel. December 2008.

[122] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[123] Michael Rieker, Jason Ansel, and Gene Cooperman. Transparent user-level checkpointing for the native posix thread library for linux. In *The 2006 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, Jun 2006.

[124] Thomas Ropars and Christine Morin. Active optimistic message logging for reliable execution of mpi applications. In *Euro-Par '09: Proceedings of the 15th International*

Euro-Par Conference on Parallel Processing, pages 615–626, Berlin, Heidelberg, 2009. Springer-Verlag.

[125] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[126] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[127] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983.

[128] G. Schneider, H. Kohlmann, and H. Bugge. Fault tolerant checkpointing solutions for clusters and grid systems, 2008.

[129] J. Schopp, K. Fraser, and M. Silbermann. Resizing memory with balloons and hotplug. 2006.

[130] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.

[131] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *in Proceedings of the 10th International Parallel Processing Symposium (IPPS '96*, pages 526–531, 1996.

[132] L. A. Stertz. Readable dirty-bits for ia64 linux ¡irs¿ internal requirements specification. Technical report, 2003.

[133] W.R. Stevens, B. Fenner, and A.M. Rudoff. *UNIX Network programming The Sockets Networking API*, volume I. Addison-Wesley, 2004.

[134] Heinz Stockinger, Flavia Donno, Erwin Laure, Shahzad Muzaffar, Peter Z. Kunszt, Giuseppe Andronico, and A. Paul Millar. Grid data management in action: Experience in running and supporting data management services in the eu datagrid project. *CoRR*, cs.DC/0306011, 2003.

[135] N. Stone, D. Simmel, T. Kielmann, and A. Merzky. An architecture for grid checkpoint and recovery services. Technical report, 2007.

[136] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[137] Gong Su and Jason Nieh. Mobile communication with virtual network address translation, 2002.

[138] Elankovan Sundararajan, Aaron Harwood, and Ramamohanarao Kotagiri. Incorporating fault tolerance with replication on very large scale grids. In *PDCAT '07: Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 319–328, Washington, DC, USA, 2007. IEEE Computer Society.

[139] SWsoft. Openvz user's guide. 2005. http://download.openvz.org/doc/OpenVZ-Users-Guide.pdf. Zuletzt besucht am 13.05.2010.

[140] Katia Sycara, Matthias Klusch, Seth Wido, and Jianguo Lu. Dynamic service matchmaking among agents in open information environments. *SIGMOD Record*, 28:47–53, 1999.

[141] Paula Ta-Shma, Guy Laden, Muli Ben-Yehuda, and Michael Factor. Virtual machine time travel using continuous data protection and checkpointing. *SIGOPS Oper. Syst. Rev.*, 42(1):127–134, 2008.

[142] Yuval Tamir and Carlo H. Séquin. Error recovery in multicomputers using global checkpoints. In *In 1984 International Conference on Parallel Processing*, pages 32–41, 1984.

[143] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[144] Geoffroy Vallee, Renaud Lottiaux, David Margery, and Christine Morin. Ghost process: a sound basis to implement process duplication, migration and checkpoint/restart in linux clusters. In *ISPDC '05: Proceedings of the The 4th International Symposium on Parallel and Distributed Computing*, pages 97–104, Washington, DC, USA, 2005. IEEE Computer Society.

[145] Werner Veith. Suns virtualbox beherrscht live-migration. Dezenber 2009. Zuletzt besucht am 13.05.2010.

[146] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in hpc environments. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[147] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[148] Darrin West and Kiran Panesar. Automatic incremental state saving. In *Proc. 10th Workshop on Parallel and Distributed Simulation (PADS 96*, pages 78–85, 1996.

[149] E. Yang, M. Artac, and A. Cernivec. Third specification of security and vo services d3.5.11, 2009.

[150] Erica Y. Yang, Brian Matthews, Amit Lakhani, Yvon Jégou, Oscar David Sánchez, Carsten Franke, Philip Robinson, Adolf Hohl, Bernd Scheuermann, Daniel Vladusic, Haiyan Yu, Rubao Lee, and Erich Focht. Virtual organization management in xtreemos: An overview, 2009.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings