Verteilte Algorithmen

Aufgabenblatt 5

(zum 28. November 2012)

Gruppe 08
Björn Stabel 222128
Tim Strehlow 316594
Friedrich Maiwald 350570

Aufgabe 5.1: Wechselseitiger Ausschluss

I*mplementieren Sie die verbesserte Token Ring-Lösung (Suzuki und Kasami, 1985) für den gegenseitigen Ausschluss mit Hilfe des Simulationsframeworks. Veranlassen Sie in (pseudo-)zufälligen Abständen einen zufällig ausgewählten Knoten dazu, die Ressource anzufordern. Wählen Sie die Zufallswerte für den Abstand sinnvoll, das heißt, es sollte sowohl Zeiten in der Simulation geben, zu denen wenige oder gar keine Anforderungen vorliegen, als auch Zeiten, zu denen sehr viele Anforderungen vorliegen.*

In unserem Programm sind normalerweise alle Knoten schwarz dargestellt. Fordert ein Knoten das Token an, wird er blau und sendet eine blaue Request-Nachricht. Das Token selber ist orange, ein Knoten mit ruhendem Token ist auch orange dargestellt. Zusätzlich zum Namen der Knoten werden auch die in ihr enthaltenen Liste der Anfragen/wartende Prozesse... angezeigt.

Aufgabe 5.2: Verteilte Speicherbereinigung

Recherchieren Sie, wie die verteilte Speicherbereinigung bei Java RMI realisiert ist und vergleichen Sie es mit dem Mechanismus, der in Microsoft DCOM eingesetzt wird.

Java RMI - Reference-Counting Algorithmus

Jeder Knoten zählt sowohl die Referenzen die er auf verteilte Objekte besitzt, als auch die Referenzen die auf ein von ihm gehaltenen verteilten Objekt bestehen.
Der Client schickt dem Server hierfür eine "referenced message" bei der ersten Referenz auf ein verteiltes Objekt. Wenn die letzte Referenz auf ein verteiltes Objekt freigegeben wird, schickt der Client dem Server eine "unreferenced message".
Gibt es keine entfernten Referenzen auf ein verteiltes Objekt mehr, so hat das RMI-Laufzeitsystem des Servers anstatt einer "normalen" Referenz bloß noch eine weak reference auf das Objekt. Gibt es nun keine lokalen Referenzen auf ein Objekt mehr und hat das RMI-Laufzeitsystem nur noch eine weak reference, so kann das Objekt vom lokalen Garbage Collector vernichtet werden.
Quelle: http://docs.oracle.com/javase/1.4.2/docs/guide/rmi/spec/rmi-arch4.html

Microsoft DCOM

Microsoft DCOM verwendet ebenso einen Reference Count Algorithmus um verteilte Speicherbereinigung umzusetzen. Darüber hinaus wird noch ein Pinging Protocol verwendet, um festzustellen ob andere Knoten oder Verbindungen zu ihnen abgestürzt sind. Somit können dann

anschließend deren Referenzen dekrementiert werden.

Der Unterschied zwischen beiden liegt in dem Pinging Protocol, mit welchem DCOM das Connection Management umgesetzt ist und wodurch auch effektivere Garbage Collection stattfinden kann, da somit Referenzen von abgestürzten Knoten nicht ewig als gegeben angesehen werden, sondern deren Count direkt dekrementiert wird. Für RMI haben wir ein vergleichbares Protokoll nicht finden können.

Aufgabe 5.3: Mark and Sweep

Das Markieren und Ausfegen ist im echt verteilten Fall ungünstig, da das System angehalten werden muss. Aber auch im nicht echt verteilten Fall (z. B. mehrere Threads auf einem PC) kann das Anhalten all dieser Threads zu einer spürbaren Verzögerung führen. Daher wird versucht, während des Markierens und Ausfegens das System nicht dauerhaft anzuhalten. Wie kann dies bewerkstelligt werden? (Siehe beispielsweise: Tony Printezis and David Detlefs: "A Generational Mostly-concurrent Garbage Collector", http://labs.oracle.com/techrep/2000/smli_tr-2000-88.pdf)

Mark-and-sweep ist die einfachste Form eines Garbage Collectors. Da die Ausführung wie in der Aufgabe beschrieben zu Verzögerungen führen kann, gibt es verschiedene Alternativvorschläge zur Lösung diese Problems.

Ein Ansatz ist das tricolor-Verfahren, bei dem Teile der Speicherbereinigung parallel zum eigentlichem Programm laufen. Zu Beginn muss dabei der Mutator (das eigentliche Programm, das Objekte im Speicher erzeugt und verändert) kurz angehalten werden, um die Root-Objekte zu markieren. Danach kann das Programm weiterlaufen und parallel dazu hangelt sich der Collector von den Root-Objekten zu allen erreichbaren (noch gebrauchten) Objekten und markiert diese. Sollte der Mutator in dieser Phase schon markierte Onjekte ändern, muss er diese für den Collector gesondert markieren. Danach wird das Programm kurz erneut gestoppt, damit der Collector die geänderten Objekte überprüfen und markieren kann. Am Ende kann das Programm weiterlaufen und parallel dazu löscht der Collector alle Objekte, zu denen keine Referenzen mehr führen. Bei diesem Verfahren können eventuell einige Objekte übersehen werden, so dass die Speicherbereinigung nicht alle nicht mehr gebrauchten Objekte erfasst. Diese werden aber beim nächsten Durchlauf des Garbage Collectors erfasst und gelöscht.

Ein andere Ansatz bezieht sich auf das Alter der Objekte. Es hat sich gezeigt, dass in durchschnittlichen Programmen die jungen Objekte meist nur kurzlebig sind und nicht lange gebraucht werden. Man kann die Objekte also in Generationen zusammenfassen und der Garbage Collector befasst sich größtenteils nur mit der jüngsten Generation. Sollte ein Objekt über mehrere Durchläufe hinweg in der jüngsten Generation überleben, wird es in eine ältere Generation "befördert" und wird dort seltener vom Garbage Collector überprüft. Mit diesem Verfahren wird die benötigte Verzögerung im Programm minimiert, da nur der Teil des Speichers überprüft wird, der am wahrscheinlichsten zu löschende Objekte enthält. Wenn dann aber auch mal die älteren Generationen vom Garbage Collector überprüft werden, hat diese Speicherbereinigung dieselben Performance-Nachteile wie einfaches Mark-and-sweep.

Das in der Aufgabe genannte Paper nennt diese beiden möglichen Verfahren zur Speicherbereinigung und entwirft einen weiteren und komplexeren Ansatz, um noch mehr

Parallelität zum Programmablauf zu erreichen.