Projektbericht

Meine Aufgabe war es, Zookeeper durch ein anderes System zu ersetzen. Zookeeper ist ein verteiltes System, das Daten konsistent und zuverlässig speichert. Die Implementierung eines solchen Speichers ist nicht trivial und enthält oft Fehler, die nicht offensichtlich sind. Deswegen wird Zookeeper von anderen verteilten Systemen benutzt, um nicht einen eigenen konsistenten Speicher zu implementieren. DXRAM benutzt Zookeeper vor allem für den Bootprozess, um sich auf eindeutige Node-IDs zu einigen und einen Bootstrap-Peer zu wählen.

1. Recherche

Zunächst habe ich überlegt, was für ein System für DXRAM benötigt wird. Dabei wurde schnell klar, dass tatsächlich ein Einigungsalgorithmus mit asynchronem Crash-Recovery-Modell wie Paxos benötigt wird, da einfachere Algorithmen wie das 2-Phasen-Commit-Protokoll in realen Systemen nicht immer funktionieren, da es keine Garantien an die Dauer der Nachrichtenübertragung gibt.

Zookeeper implementiert einen Einigungsalgorithmus namens Zab (Junqueira, 2011). Zab ist dem bekannten Paxos-Algorithmus ähnlich, bietet jedoch weitergehende Garantien an die Reihenfolge der Zustandsänderungen. Paxos implementiert eine Replicated State Machine, wobei sich die Reihenfolge von einmal angelegten Log-Einträgen noch verändern kann, falls der Leader ausfällt, solange sie nicht committed wurden. Bei Zab muss jedoch die Reihenfolge der Einträge immer der Reihenfolge entsprechen, in der sie angelegt wurden. Das nennen die Autoren ein „primary-backup system“. Dieses Feature sollte für den vorgesehen Einsatz in DXRAM jedoch nicht wichtig sein.

Bei der Suche nach vorhandenen Paxos-Implementierungen, habe ich noch den Raft-Algorithmus (Raft Consensus Algorithm, 2018), (Ongaro, 2014) entdeckt, der ebenfalls eine Replicated State Machine umsetzt, aber laut den Autoren einfacher zu verstehen ist. Außerdem habe ich von Raft wesentlich mehr Implementierungen gefunden als von Paxos, was darauf schließen lässt, dass Raft tatsächlich einfacher zu implementieren ist als Paxos. Außerdem konnte ich wesentlich ausführlichere und implementierungsnähere Informationen zu Raft finden.

Die vorhandenen Raft-Implementierungen, insbesondere die Java-Implementierungen, waren jedoch entweder nicht vollständig oder nicht ohne Aufwand flexibel genug für die vorgesehene Verwendung von DXNet. Deswegen habe ich mich entschieden, eine eigene Raft-Implementierung zu entwickeln, insbesondere auch im Hinblick auf eventuelle Performance-Optimierungen speziell für den Einsatz in DXRAM, die mit einem eigenen System später vermutlich besser vorgenommen werden können.

2. DXRaft

DXRaft sollte eine leichtgewichtige und zuverlässige Raft-Implementierung werden. Bei der Entwicklung habe ich festgestellt, dass zunächst der Fokus auf der Zuverlässigkeit liegen sollte, da es oft schwierig ist, alle Fälle zu bedenken und Fehler oft nicht direkt erkennbar sind. Deswegen habe

ich auch einige (Unit-)Tests geschrieben und den Fokus zunächst nicht auf die Performance oder den Funktionsumfang gelegt.

DXRaft sollte außerdem nicht nur standalone benutzt werden können, sondern auch als Library in vorhandene Systeme eingebunden werden können. Außerdem sollte die Netzwerkschicht relativ einfach ausgetauscht werden können.

Eine Server-Instanz kann standalone mit der main-Methode in der DXRaft-Klasse oder als Library mit einer Instanz der Klasse RaftServer gestartet werden. Dabei muss die vorgesehene Cluster-Konfiguration mit IDs und Adressen übergeben werden. Optional kann auch eine Implementierung des Network-Service-Interfaces übergeben werden. Standardmäßig wird ein einfacher UDP-basierter Network-Service mit einem einzelnen Listening-Thread, der auch das Verarbeiten der Nachrichten übernimmt, genutzt. Ein weiterer (Timer-)Thread startet, wenn eine Weile keine Nachrichten empfangen werden und handelt entsprechend dem aktuellen Zustand des Servers.

Ein Client kann ebenfalls standalone ausgeführt werden, mit eine main-Methode in der RaftClient-Klasse oder es kann eine Instanz dieser Klasse angelegt werden. Dabei muss die Adresse von mindestens einem Server des Clusters übergeben werden, besser jedoch alle, da sonst das Cluster nicht mehr vom Client erreichbar ist, falls der entsprechende Server ausfällt. Mit dem Client können Daten geschrieben, gelesen, gelöscht und atomar verglichen und geschrieben werden (mit einer compareAndSet-Methode). Außerdem können Listen angelegt werden, zu denen Daten hinzugefügt und wieder entfernt werden können. Die Daten werden mit einem Namen-String identifiziert, die Datenstruktur ist also eine einfache Map.

Über den Client können ebenfalls Server dem Cluster hinzugefügt und entfernt werden. Noch nicht implementiert ist die Möglichkeit, dass abgestürzte Server beim erneuten Start der Anwendung automatisch dem Cluster wieder beitreten. Das heißt, wenn ein Server abstürzt muss er erst per Konfigurationsänderung aus dem Cluster entfernt werden und dann wieder hinzugefügt werden. Dabei wird der Server dann auf den aktuellen Zustand gebracht.

Außerdem sind Read-Anfragen im Moment genauso implementiert, wie Schreibanfragen, das heißt sie durchlaufen den gleichen Replikationsprozess und werden im Log gespeichert. Da diese den Zustand jedoch nicht verändert, kann man Read-Anfragen auch so implementieren, dass diese nicht im Log gespeichert werden, was das Lesen deutlich schneller machen würde. Dafür muss man allerdings einige Dinge beachten, damit nicht alte Daten gelesen werden. Dazu gibt es auch ein Kapitel in (Ongaro, 2014).

Des Weiteren sollte das System auch noch ausführlicher getestet werden und eventuell auch noch mehr Unit Test hinzugefügt werden, um sicher zu gehen, dass das System zuverlässig arbeitet.

3. Integration in DXRAM

Zunächst habe ich versucht, DXRaft direkt in DXRAM zu integrieren, sodass beides im gleichen Prozess läuft. Leider bin ich dabei auf Problemen gestoßen, z.B. benötigen die Raft-Server ebenfalls IDs, um sich gegenseitig zu identifizieren. Deswegen ist es nicht möglich, Raft für die ID-Vergabe in DXRAM zu benutzen. Außerdem kann auch DXNet nicht benutzt werden, bevor der Knoten eine ID

besitzt. Da ich dafür keine sinnvolle Lösung gefunden habe, habe ich erstmal nur die Client-Library in DXRAM integriert.

DXRaft muss deswegen vorher standalone gestartet werden. Dann muss DXRAM eine Liste mit Adressen mitgegeben werden, unter denen die Raft-Server laufen. Dazu habe ich die Klasse *DXRaftBootComponent* hinzugefügt, die sich analog zu *ZookeeperBootComponent* mit DXRaft verbindet und die Vergabe der IDs und die Wahl eines Bootstrap-Peers übernimmt.

Literaturverzeichnis

Junqueira, R. S. (2011). Zab: High-performance broadcast for primary-backup systems. *2011 IEEE/IFIP 41st International Conference on Dependable System & Networks (DSN)*, (S. 245-256).

Ongaro, D. (2014). *Consensus: Bridging Theory and Practice.* Stanford University.

Raft Consensus Algorithm. (22. August 2018). Von https://raft.github.io. abgerufen