# Überblick

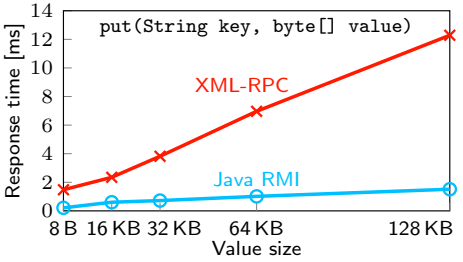

**Effizienz**

- Motivation
- Anwendungsbasierte Ansätze
- Asynchrone Fernaufrufe
- Remote Evaluation
- RDMA-basierte Fernaufrufe




# Motivation

- Interaktionen im verteilten System **aufwändiger als im lokalen Fall**
  - Kommunikation über ein Netzwerk
  - Erstellung und Verarbeitung von Nachrichten
- Beispiel: Einfügeoperation eines Speichers für Schlüssel-Wert-Paare
  - Latenz bei lokalem Methodenaufruf: $< 1\,\mu s$
  - Evaluierung im verteilten Fall (lokales Netzwerk)




- Herausforderungen
  - Wie lässt sich die **Latenz eines Fernaufrufs reduzieren**?
    - Auf Applikationsebene
    - Unterstützt durch die Hardware
  - Wie kann man die im Netzwerk bedingten **Verzögerungen verschatten**?

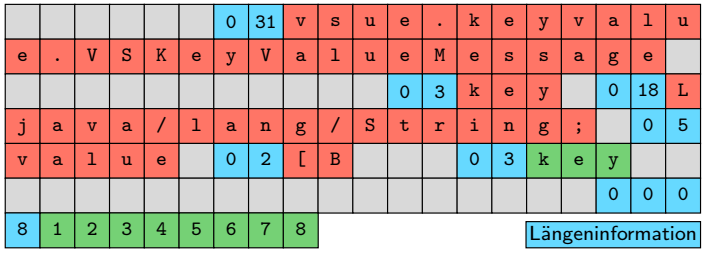


# Optimierte Serialisierung — Einsparpotenzial in Java

- Beispielklasse (Package `vsue.keyvalue`)

```
public class VSKeyValueMessage implements Serializable {
  private String key;
  private byte[] value;
}
```

- Objektserialisierung per `ObjectOutputStream`
  - Beispiel: `key` = „key“ und `value` = `new byte[] {1,2,3,4,5,6,7,8}`
  - **Anteil der Nutzdaten**: 11 von 129 Bytes (8,5%)
  - Anteil von Zeichenketten mit Strukturinformationen: 59 Bytes (45,7%)






# Optimierte Serialisierung — Vorgehensweise

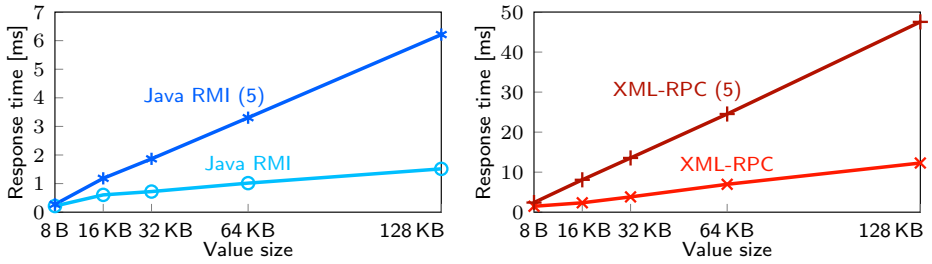
- Ansätze zur Reduzierung von Nachrichtengrößen
  - Fokus auf Übertragung von **Informationen, die dem Empfänger fehlen**
  - Weglassen von Wissen, das der Empfänger bereits besitzt

→ Einsatz von Anwendungswissen

- Beispiele
  - Ausnutzung von **statischen Strukturinformationen**
    - Nachrichtentypen
    - Nachrichtenaufbau
    - Datentypen
  - Kompakte Repräsentation von Methoden in Java RMI
    - Austausch eines Hash über den Methodennamen und -deskriptor
    - Caching von Methoden-Hashes
  - Einsparungen bei mehrfacher Übertragung derselben Objekte
    - Weitergabe einer vollständigen Kopie beim ersten Senden eines Objekts
    - Übermittlung von Änderungen bei nachfolgenden Sendeaufrufen
  - Verwendung von Kenntnissen über **eingeschränkte Wertebereiche**

# Bündelung von Aufrufen auf Anwendungsebene

- Kosten pro Fernaufruf
  - Daten: Nachrichten-Header für Anfragen und Antworten
  - Kommunikation: Sende- und Empfangsoperationen
- Ansatz zur Kostenreduzierung: **Bündelung von Aufrufen** (*Batching*)
- Beispiel: Einfügeoperation für mehrere Schlüssel-Wert-Paare
  - Methode: `putAll(String[] keys, byte[][] values)`
  - Beobachtung: **Effizienzgrad ist abhängig von Nachrichtengröße**
  - Evaluierung für eine Batch-Größe von 5






# Asynchrone Fernaufrufe

- Grundprinzip
  - Fernaufruf kehrt sofort zurück
  - Rückgabe eines **Promise-Objekts**
- *Promise* [Vergleiche: `Futures` in Java.]
  - **Container für das Ergebnis** eines Fernaufrufs
  - Zustände
    - *Blockiert* Resultat liegt noch nicht lokal vor
    - *Bereit* Ergebnis ist bereits bei Aufrufer eingetroffen
  - Operationen
    - `ready()` Abfrage des Zustands
    - `claim()` Abholen des Resultats (blockiert gegebenenfalls)
- Literatur

Barbara Liskov and Liuba Shrira
**Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems**
*Proceedings of the 9th Conference on Programming Language Design and Implementation (PLDI '88)*, S. 260–267, 1988.



# Spekulative Ausführung

- Asynchrone Fernaufrufe
  - Vorteile gegenüber synchronen Fernaufrufen
    - Aufrufer kann während der Ausführung eines Fernaufrufs weiterarbeiten
    - Blockade erst, sobald ohne Fernaufrufresultat kein Fortschritt möglich
    - Optimal, falls Aufrufer nicht am Ergebnis interessiert ist
  - Problemszenario: Sequenz **voneinander abhängiger Fernaufrufe**
- Spekulative Ausführung
  - Grundidee: Effizienzsteigerung durch **Vorhersage zukünftiger Ergebnisse**
  - Vorgehensweise
    1. Initiierung eines asynchronen Fernaufrufs
    2. Erstellen eines lokalen Sicherungspunkts
    3. Fortsetzung mit dem wahrscheinlichsten Ergebnis
    4. Auswertung des Ergebnisses bei dessen Eintreffen
       - Verwerfen des Sicherungspunkts bei korrekter Vorhersage
       - Sonst: Zurücksetzen des Aufrufers auf den Sicherungspunkt (*Rollback*)
  - **Bedingung für Konsistenz**: Keine Externalisierung spekulativen Zustands



# Speculator Überblick

- Spekulative Ausführung in verteilten Dateisystemen (z. B. NFS)
  - Beobachtungen
    - Ausgang von Operationen sind vom Client relativ **zuverlässig vorhersagbar**
    - Effiziente Erzeugung von Sicherungspunkten für Prozesse möglich
    - Akzeptabler Ressourcenmehraufwand für spekulative Ausführung
  - Hohe Effektivität in weitverteilten Systemen
- *Speculator*
  - **Betriebssystemunterstützung** für spekulative Ausführung
  - Erstellen von Sicherungspunkten für Prozesse
  - Zurücksetzen und Neuausführung von Prozessen
- Literatur

Edmund B. Nightingale, Peter M. Chen, and Jason Flinn
**Speculative execution in a distributed file system**
*ACM Transactions on Computer Systems*, 24(4):361–392, 2006.

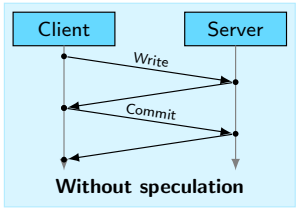
## Speculator — Ausgangsbasis

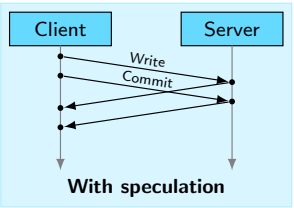
- NFSv3-System
  - Server
    - Persistente Speicherung von Dateien auf der Festplatte
    - **Zentraler Synchronisationspunkt** des Systems
  - Clients
    - Verwaltung eines lokalen Datei-Cache
    - Interaktion mit dem Server erfolgt per Fernaufruf (Beispiele)
      - `getattr()` Auslesen von Dateiattributen
      - `write(UNSTABLE)` Übertragen eines Datenblocks auf den Server
      - `commit()` Schreiben der übertragenen Blöcke einer Datei
- *Close-to-Open*-Konsistenz
  - Nach dem Öffnen einer Datei sind alle Modifikationen von anderen Clients an dieser Datei nur sichtbar, falls diese die Datei **zuvor geschlossen** haben
  - Implementierung auf Client-Seite ohne spekulative Ausführung
    - Beim Schließen: **Asynchrones Hochladen** lokaler Änderungen per `write()`
    - Anschließend: Persistente Speicherung mittels synchronem `commit()`
    - Beim Öffnen einer gecachten Datei: Prüfung auf Modifikation (`getattr()`)



## Speculator — Spekulative Ausführung

- Beim Öffnen einer gecachten Datei
  - Spekulation: Dateiversion im lokalen Cache ist aktuell
  - Nach fehlerhafter Prognose: **Invalidierung** der betroffenen Cache-Einträge
- Beim Schließen einer Datei
  - Spekulation: Übertragung der geänderten Datenblöcke ist erfolgreich
  - Asynchroner Aufruf von `commit()` noch vor Abschluss aller `write()`s
    - Identifizierung von Operationen mittels **Sequenznummern**
    - `commit()`-Anfrage enthält Sequenznummern aller abhängigen `write()`s
    - Server verwaltet Client-spezifische **Liste mit fehlgeschlagenen Operationen**
    - Ausführung von `commit()` erfolgt nur, falls alle `write()`s erfolgreich waren








## Remote Evaluation

- **Austausch von Programmen** statt der Daten, auf denen sie arbeiten
- Chancen
  - Entlastung des Netzwerks bei großen Mengen zu verarbeitender Daten
  - Erweiterung der Funktionalität eines Diensts zur Laufzeit möglich
  - **Verlagerung aufwändiger Operationen** auf leistungsstärkere Rechner
  - Reduzierung der Anzahl für komplexe Aufgaben erforderlicher Fernaufrufe
- Risiko: Verlust der Systemstabilität
  - Beeinträchtigung der Sicherheit
  - Gefährdung von Dienstgütegarantien

→ Konsequenzen für den Einsatz in der Praxis

  - Beschränkung auf vertrauenswürdige Umgebungen und/oder
  - Bereitstellung **effektiver Schutzmechanismen**
- Literatur

James W. Stamos and David K. Gifford
**Remote evaluation**
*ACM Trans. on Programming Languages and Systems*, 12(4):537–564, 1990.



## Extensible ZooKeeper — Überblick

- ZooKeeper [Nähere Details in der Vorlesung „*Middleware – Cloud Computing*“.]
  - Von Yahoo entwickelter, Java-basierter Koordinierungsdienst
  - Um Zusatzfunktionen erweiterter **Speicher für Schlüssel-Wert-Paare**
  - Verwaltung eines Versionszählers pro Schlüssel-Wert-Paar
- Extensible ZooKeeper
  - Dynamisches **Hinzufügen von Funktionalität** per *Erweiterung (Extension)*
    - Aktivierung: Schreiben des globalen Schlüssels `ezk` [`put(ezk,<Erweiterungs-Code>)`]
    - Kompilieren der Erweiterung auf Server-Seite
    - Ausführung einer Erweiterung durch Zugriff auf ihren Schlüssel *s* [z. B. `get(s)`]
  - **Schutzmechanismen** auf Server-Seite
    - Statische Analyse des Erweiterungs-Codes
    - Ausführung von Erweiterungen in Sandboxes
- Literatur



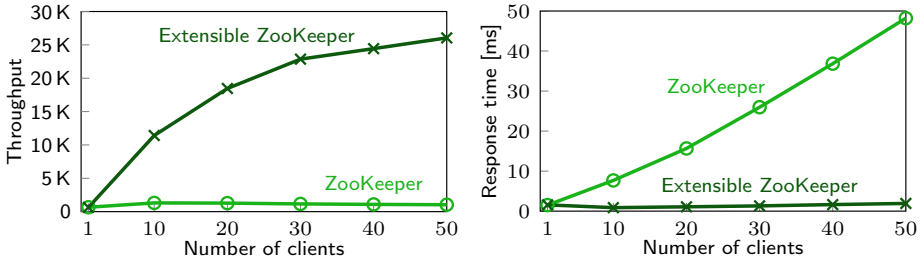
Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer et al.
**Extensible distributed coordination**
*Proceedings of the 10th European Conference on Computer Systems (EuroSys '15)*, S. 143–158, 2015.

## Extensible ZooKeeper — Beispiel

- Inkrementieren des Zählers `key` in ZooKeeper (Client-Pseudo-Code)

```
ZooKeeper zk = [Verbindungsaufbau zum ZooKeeper-Dienst];
do {
  [value, version] = zk.get(key);
  zk.put(key, value + 1, version);      // Wirft Exception, falls version veraltet
} while(Aufruf von put() gescheitert);
```

- Extensible ZooKeeper: Inkrementoperation ist Teil von `zk.get(key)`
  - **Atomares Erhöhen des Zählers** durch die Erweiterung auf Server-Seite
  - Keine Konfliktbehandlung (`while`-Schleife) erforderlich
- Evaluierung






## RDMA-basierte Fernaufrufe — Überblick

- *Remote Direct Memory Access (RDMA)*
  - **Hardware-unterstützter Zugriff** auf den Speicher eines anderen Rechners
  - Direkte Interaktion mit der Netzwerkkarte
  - Umgehung des Betriebssystems beim Sender und Empfänger
  - **Kein Umkopieren** auf unteren Schichten nötig (*Zero-Copy Networking*)
  - Beispiele: RDMA over Converged Ethernet, InfiniBand, iWARP
- *Data Center RPC*
  - Java-basiertes Fernaufrufsystem (IBM SDK)
  - **RDMA-Netzwerk-Stack auf Nutzerebene**
  - Wiederverwendung von Datenstrukturen bei mehrmaligen Anfragen
  - 2,4 Millionen Fernaufrufe/s bei einer Latenz von 10 $\mu$s
- Literatur

Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle
**DaRPC: Data center RPC**
*Proceedings of the 5th Symposium on Cloud Computing (SoCC '14)*, 2014.



## RDMA-basierte Fernaufrufe — Funktionsweise

- Kommunikation mit der Netzwerkkarte mittels **Warteschlangen**
  - *Send-Queue* für Sendeaufträge
  - *Receive-Queue* für Empfangsaufträge
  - *Completion-Queue* für Bestätigungen ausgeführter Aufträge
- **Vorgehensweise bei Fernaufruf**
  1. Skeleton stellt Empfangspuffer bereit und gibt Empfangsauftrag
  2. Stub erstellt Anfrage und schreibt sie in einen Sendepuffer
  3. Netzwerkkarte überträgt den Inhalt des Sendepuffers zum Server
  4. Netzwerkkarte des Servers schreibt die Daten in den Empfangspuffer
  5. Skeleton erhält Empfangsbestätigung und deserialisiert die Anfrage
  6. Skeleton führt die Anfrage aus und sendet eine Antwort
- Optimierungen auf Server-Seite
  - **Gemeinsame Warteschlangen** für verschiedene Verbindungen
  - Lastbalancierung zwischen Warteschlangen
  - Einsatz mehrerer Netzwerkkarten