# Einsatz von Git + Git LFS als Client-Server-Transportsystem im Vergleich zu rsync

Dieses Papier stellt dar, warum Git in Kombination mit Git LFS ein leistungsfähigeres, strukturierteres und zuverlässigeres Client-Server-Transportsystem sein kann als ein klassisches rsync-basiertes Setup. Dabei werden die funktionalen, technischen und betrieblichen Vorteile klar herausgearbeitet.

---

## 1. Einleitung

Traditionell wird rsync verwendet, um Dateien zwischen Clients und einem Server zu übertragen. rsync ist schnell, effizient und bewährt – jedoch arbeitet es ausschließlich dateibasiert, ohne Versions- oder Zustandsmodell. Für moderne, verteilte Upload-Systeme mit vielen Clients, großen Binärdaten und hohen Anforderungen an Nachvollziehbarkeit und Fehlertoleranz stößt rsync an seine Grenzen.

Git in Kombination mit Git LFS bietet eine leistungsfähige Alternative, die nicht nur funktional überlegen ist, sondern auch strukturelle Vorteile bringt.

---

## 2. Grundlegende Unterschiede

### rsync

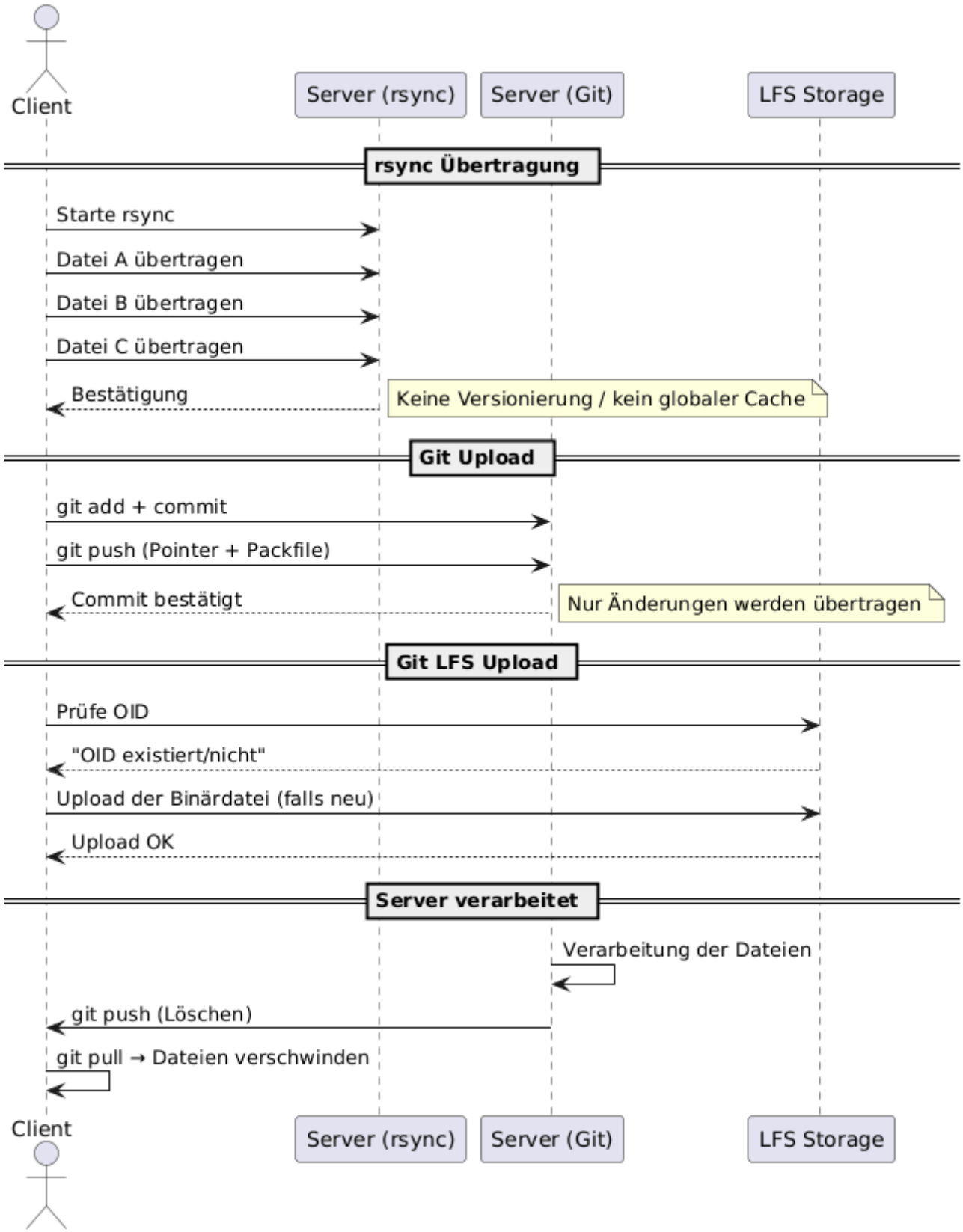
- Arbeitet dateibasiert
- Kein integriertes Versionsmanagement
- Keine globale Deduplizierung
- Keine Rückmeldung über verarbeitete Dateien
- Löschen und Aufräumen muss manuell implementiert werden
- Kein systemweiter Zustand – nur Datei für Datei

### Git + Git LFS

- Versionskontrolle, Historie, Commit- und Branch-Struktur
- Globale inhaltsbasierte Deduplizierung (SHA-basierte OIDs)
- Automatisches Synchronisieren von Änderungen und Löschungen
- Speicherbereinigung über Git-GC und LFS-Prune
- Perfekte Nachvollziehbarkeit durch Log- und Commit-Historie
- Strukturiertes, konsistentes System statt losem Dateiablageort

---

## 3. Vorteile von Git + Git LFS gegenüber rsync




### 3.1 Effizientere Übertragung vieler kleiner Dateien

Git bündelt alle Änderungen in einem komprimierten Packfile. Dadurch werden viele kleine Textdateien effizienter übertragen als bei rsync, das für jede Datei Separatoperationen ausführt.

rsync

```
Client               Server
  |                    |
  |--- Datei A -------> |   (immer vollständiger Transfer)
  |--- Datei B -------> |
  |--- Datei C -------> |
  |                    |
  |--- Datei X -------> |   (auch wenn identisch!)
```

**Eigenschaften:**

- Kopiert jede Datei, auch wenn ein anderer Client sie bereits geschickt hat.
- Keine inhaltsbasierte Deduplizierung.
- Keine Versionierung.
- Für den gebündelten Transfer muss gzip, zlib oder andere Komprimierungstechnologien verwendet werden.

---

### Git (nur Textdateien)

```
Client                        Server
  |                             |
  |------ packfile ------------->|   (ein komprimiertes Paket)
  |                             |
  |  neue Blobs werden übertragen|
  |  alte Blobs → kein Transfer  |
```

**Eigenschaften:**

- Extrem effizient für viele kleine Dateien.
- Starke Kompression.
- Versionierung durch Commits.

## 3.2 Globale Deduplizierung großer Binärdateien

Mit Git LFS werden Binärdateien per SHA-256 identifiziert. Hat der Server eine Datei bereits, wird sie **kein zweites Mal** übertragen. Bei rsync würden gleiche Dateien von jedem Client erneut übertragen.

### Git LFS (für WAV/Binärdaten)

```
Client                                Server
  |                                     |
  |-- Git: Pointerdatei push ----------> |
  |                                     |
  |-- LFS: Upload OID Anfrage ----------> |
  |<-- LFS: "OID existiert nicht" ------- |
  |-- LFS: Binärdaten (Chunked) --------> |
```

```
  |                                        |
  |<-- LFS: "Upload erfolgreich" -------- |
```

Falls Datei bereits existiert:

```
Client                                Server
  |                                     |
  |-- Git: Pointerdatei push ----------> |
  |-- LFS: Upload OID Anfrage ----------> |
  |<-- LFS: "OID existiert bereits" ------|
  |          -> 0 Byte Upload             |
```

**Eigenschaften:**

- Globale Deduplizierung über SHA-256.
- Mehrere Clients → identische Datei wird nur 1x übertragen.

### 3.3 Automatisches Löschen auf dem Client

Wird eine Datei auf dem Server verarbeitet und per Git entfernt, verschwindet sie beim nächsten `git pull` automatisch auch auf dem Client. Ein rsync-System kann diesen Synchronisationsmechanismus nicht ohne zusätzliche Logik nachbilden.

#### rsync

```
Client             Server
  |                  |
  |  Datei löschen   |
  |------X---------->|
                     |   (Server löscht nicht automatisch)
                     |   kein globaler Zustand
```

#### Git / Git LFS

```
Server führt Verarbeitung durch
      |
      | git rm Datei
      | git commit
      | git push
      v
Client
      | git pull
      v
Datei verschwindet automatisch
```

**Eigenschaft:** Global konsistente Löschung über Versionskontrolle.

### 3.4 Versionshistorie und forensische Nachvollziehbarkeit

Git protokolliert:

- wann Dateien eingespielt wurden
- wann sie gelöscht wurden
- durch welchen Commit eine Verarbeitung stattfand

Dies ermöglicht eine perfekte Korrelation mit Systemlogs und erleichtert Debugging, Fehleranalyse und Compliance stark.

### 3.5 Strukturierte Upload-Prozesse über Branches

Statt flacher Ordnerstrukturen können Uploads sauber über Branches organisiert werden (z. B. `upload/clientA/batch123`). Diese Branches definieren isolierte Zustände, in denen der Server verarbeiten und danach aufräumen kann.

### 3.6 Kontrolliertes Aufräumen durch Git

Nicht mehr benötigte Daten werden auf dem Server durch:

- `git rm`
- `git commit`
- `git push`
- `git gc`
- `git lfs prune`

vollständig und konsistent entfernt – inklusive auf allen Clients.

### 3.7 Keine doppelten Dateien im Repo

Git LFS speichert große Dateien **nicht** automatisch doppelt. Das Working Directory enthält die Originaldatei, im Git-Repo selbst wird nur ein Pointer gespeichert, wodurch kaum zusätzlicher lokaler Speicher verbraucht wird.

### 3.8 Sicherer Transfer über Commit-basierte Bestätigung

Der Server kann erst dann löschen, wenn ein Commit signalisiert, dass die Verarbeitung erfolgreich war. So entsteht ein robustes, fehlertolerantes System ohne Risiko von Datenverlust.

### 3.9 Direkter Streaming-Upload ohne lokale Zwischenspeicherung (JGit + LFS)

Moderne Java-Implementierungen wie Eclipse JGit und kombinierbare Git-LFS-Bibliotheken erlauben es, große Dateien direkt im Stream von Client zu Server zu übertragen, ohne sie vollständig lokal abzulegen. Dadurch entsteht eine hochmoderne, ressourcenschonende Architektur mit folgenden Vorteilen:

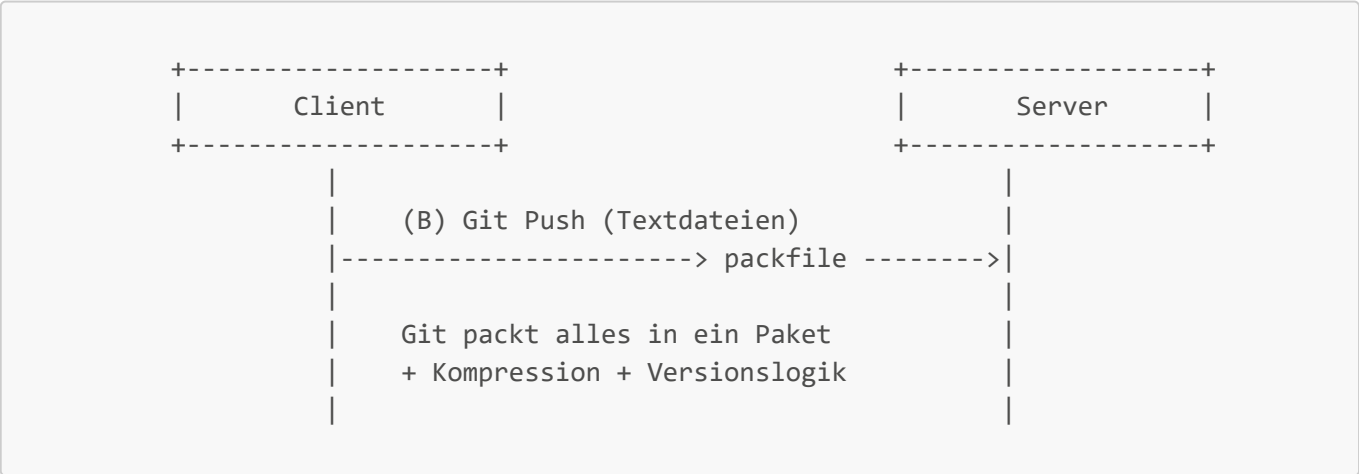
- **Kein doppelter Speicherverbrauch**: Große Binärdaten müssen nicht zuerst lokal komplett gespeichert und dann an den Server übertragen werden.

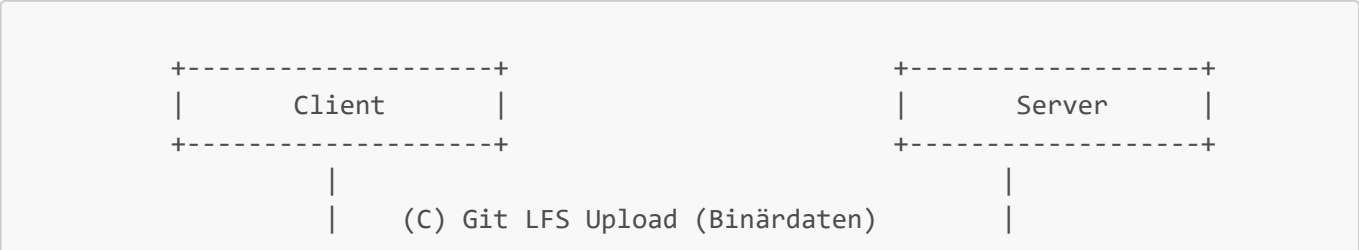
- **Weniger Risiko lokaler Datenkorruption**: Das Fehlen lokaler Zwischenschritte reduziert typische Fehler wie unvollständige Downloads/Uploads oder Dateisystemprobleme.
- **Effizientes Arbeiten mit großen Datenmengen**: Streaming liefert direkte Weitergabe der Daten an den LFS-Server, ohne Disk-I/O-Overhead.
- **Commit- & Hash-validierte Übertragung**: Die Integrität der Daten wird unmittelbar über Git-Objekthashes (Commit-, Tree- oder OID-Hash) abgesichert.
- **Natürliche Backpressure**: Der Client kann nur so schnell streamen, wie der Server bereit ist zu empfangen, was die Stabilität in Systemen mit niedriger Bandbreite erhöht.
- **Pipeline-Verarbeitung möglich**: Während gestreamt wird, können Daten bereits serverseitig verarbeitet, analysiert oder in weitere Systeme weitergeschoben werden.
- **Ideal für hochskalierbare Upload-Systeme**: Gerade bei vielen Clients und großen Dateien sorgt Streaming für minimale lokale Systemlast.

Dieser Ansatz ist dem klassischen rsync-Verhalten deutlich überlegen, da rsync immer auf vollständige lokale Dateien angewiesen ist und weder Streaming, noch Backpressure, noch Hash-basierte Git-Integrität bietet.

---

# 4. Zusammenfassung

```
                    |-----------------> Pointer per Git ------->|
                    |-----------------> Binaries per LFS ------>|
                    |                                           |
                    |  LFS prüft: "Kenn ich die Datei schon?"   |
                    |        → Ja: 0 Byte Upload                |
                    |        → Nein: vollständiger Upload       |
                    |                                           |
```

Git-basierte Upload- und Verarbeitungssysteme bieten gegenüber rsync deutliche Vorteile, vor allem in komplexeren Szenarien mit verteilten Clients und großen Binärdateien. Git + Git LFS ermöglicht nicht nur reines Dateiübertragen, sondern:

- konsistente, versionskontrollierte Upload-Pipelines
- automatische Synchronisation von Änderungen und Löschungen
- globale Deduplizierung großer Dateien
- sauberes Speicher- und Lifecycle-Management
- forensische und auditierbare Historien
- integrierte Sicherheit über Hashing und Commit-Strukturen

Diese Eigenschaften machen Git + Git LFS zu mehr als nur einem Ersatz für rsync – es ist ein technologischer Fortschritt. Bei einfachen Synchronisationen zwischen zwei Rechnern ist rsync sicherlich schnell, unkompliziert und ressourcenschonend. Doch in verteilten Systemen mit vielen Clients, bei denen große Binärdateien häufig und redundant übertragen werden, zeigen sich die Stärken von Git + Git LFS besonders deutlich. Durch die Versionskontrolle, Deduplizierung und intelligentes Netzwerk-Caching ist es die leistungsfähigere und fehlertolerantere Lösung, die strukturiert, automatisiert und nachvollziehbar arbeitet. Allerdings erfordert Git LFS in großen Umgebungen eine gute Konfiguration, insbesondere hinsichtlich Batch-Downloads, Caching und paralleler Übertragung, um seine Vorteile voll auszuspielen. Ohne diese Optimierungen kann es in bestimmten Fällen langsamer sein als ein gut eingestelltes rsync, das in einfachen Netzwerken und bei wenig geänderten Dateien punktet. Insgesamt empfiehlt sich Git + Git LFS vor allem dort, wo neben effizientem Datei-Transfer weitere Anforderungen an Versionierung, automatisierte Löschung, auditierbare Historien und Speicheroptimierung bestehen. Das macht es zur besseren Wahl für moderne, verteilte Client-Server-Uploadsysteme mit hohem Anspruch – while rsync für einfache, punktuelle Synchronisationen weiterhin ein pragmatisches Werkzeug bleibt. Dieser kombinierte Ansatz, der die Vorteile beider Tools in den Kontext der jeweiligen Einsatzszenarien stellt, bietet eine fundierte Entscheidungsgrundlage für den Technologiewechsel in Upload- und Synchronisationsinfrastrukturen

---

# 5. Empfehlung

Wenn:

- viele Clients Daten liefern,
- große Binärdateien beteiligt sind,
- ein „OK zum Löschen“-Mechanismus benötigt wird,
- Versionierung oder Auditing wichtig ist,
- oder Speicher optimiert werden muss,
- keine Zwischenspeicherung erforderlich ist,

dann ist Git + Git LFS dem klassischen rsync klar überlegen und sollte bevorzugt eingesetzt werden.