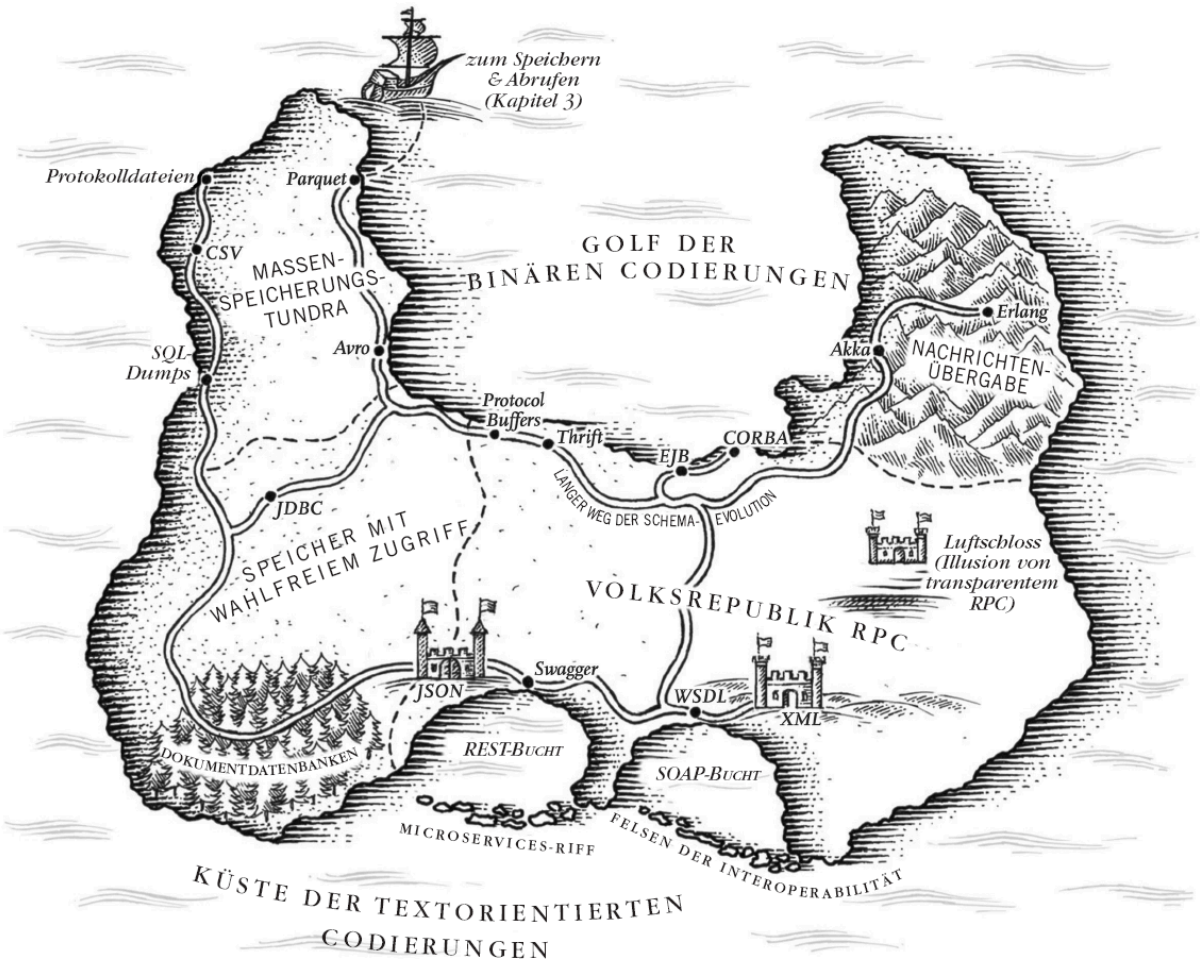
O'REILLY®

# Datenintensive Anwendungen designen

KONZEPTE FÜR ZUVERLÄSSIGE, SKALIERBARE UND WARTBARE SYSTEME

Martin Kleppmann
Übersetzung von Frank Langenau

# Inhalt




KAPITEL 4

# Codierung und Evolution

*Alles bewegt sich fort und nichts bleibt*.

– **Heraklit von Ephesus**, wie von Platon zitiert im *Kratylos* (360 v. Chr.)

Anwendungen ändern sich mit der Zeit zwangsläufig. Features werden hinzugefügt oder modifiziert, wenn neue Produkte auf den Markt kommen, den Anforderungen der Benutzer besser entsprochen werden soll oder sich die geschäftlichen Verhältnisse ändern. Kapitel 1 hat das Konzept der

*Evolvierbarkeit* eingeführt: Wir sollten danach streben, Systeme zu erstellen, die sich leicht an Veränderungen anpassen lassen (siehe Abschnitt »Evolvierbarkeit: Änderungen erleichtern« auf Seite 23).

In den meisten Fällen sind bei einer Änderung der Features einer Anwendung auch die Daten zu ändern, die die Anwendung speichert: Vielleicht müssen Sie ein neues Feld oder einen Datensatztyp erfassen oder vorhandene Daten auf eine neue Art und Weise darstellen.

Die in Kapitel 2 beschriebenen Datenmodelle haben verschiedene Möglichkeiten, um mit derartigen Änderungen klarzukommen. Relationale Datenbanken gehen im Allgemeinen davon aus, dass alle Daten in der Datenbank einem Schema entsprechen: Obwohl dieses Schema geändert werden kann (über Schemamigrationen, d.h. `ALTER`-Anweisungen), ist zu jedem Zeitpunkt genau ein Schema in Kraft. Da im Unterschied dazu Schema-on-read-Datenbanken (»schemalose Datenbanken«) kein Schema erzwingen, kann die Datenbank eine Mischung aus älteren und neuen Datenformaten enthalten, die zu verschiedenen Zeitpunkten geschrieben wurden (siehe Abschnitt »Schemaflexibilität im Dokumentmodell« in Kapitel 2 auf Seite 42).

Wenn sich ein Datenformat oder Schema ändert, ist oftmals eine entsprechende Änderung am Anwendungscode erforderlich (zum Beispiel fügen Sie ein neues Feld in einen Datensatz ein, und der Anwendungscode liest und schreibt dieses Feld). Allerdings ist es in einer großen Anwendung oftmals nicht möglich, Codeänderungen an einem einzigen Zeitpunkt in Kraft treten zu lassen:

- Bei serverseitigen Anwendungen kann es vorteilhaft sein, ein *Rolling Upgrade* (auch als *mehrphasige Bereitstellung* bezeichnet) durchzuführen: das heißt, die neue Version auf einigen Knoten auf einmal bereitstellen; sich vergewissern, ob die neue Version ordnungsgemäß läuft; und dann nach und nach die anderen Knoten aktualisieren. Da sich dadurch neue Versionen ohne Dienstausfälle bereitstellen lassen, fördert diese Vorgehensweise häufigere Releases und bessere Evolvierbarkeit.
- Bei clientseitigen Anwendungen sind Sie vom Benutzer abhängig, der das Update vielleicht nicht sofort installiert.

Das heißt, dass alte und neue Versionen des Codes sowie alte und neue Datenformate möglicherweise gleichzeitig im System existieren. Damit das System weiterhin ordnungsgemäß läuft, müssen Sie Kompatibilität in beide Richtungen aufrechterhalten:

*Abwärtskompatibilität*

Neuerer Code kann Daten lesen, die von älterem Code geschrieben wurden.

*Vorwärtskompatibilität*

Älterer Code kann Daten lesen, die von neuerem Code geschrieben wurden.

Abwärtskompatibilität ist normalerweise nicht schwer zu erreichen: Als Autor des neueren Codes kennen Sie das Format der Daten, die mit dem älteren Code geschrieben wurden, und Sie können diese explizit verarbeiten (indem Sie gegebenenfalls einfach den alten Code beibehalten, um die alten Daten zu lesen). Vorwärtskompatibilität kann diffiziler sein, weil sie voraussetzt, dass älterer Code die Erweiterungen ignoriert, die eine neuere Version des Codes vorgenommen hat.

In diesem Kapitel sehen wir uns mehrere Formate an, um Daten zu codieren, einschließlich JSON, XML, Protocol Buffers, Thrift und Avro. Insbesondere gehen wir darauf ein, wie sie Schemaänderungen behandeln und wie sie Systeme unterstützen, bei denen alte und neue Daten sowie alter und neuer Code gemeinsam existieren müssen. Dann erläutern wir, wie diese Formate für das Speichern von Daten und für die Kommunikation verwendet werden: in Webservices, REST (Representational State Transfer) und RPCs (Remote Procedure Calls) sowie in Systemen für den Nachrichtenaustausch wie zum Beispiel Aktoren und Nachrichtenwarteschlangen.

# Formate für das Codieren von Daten

Programme arbeiten üblicherweise mit Daten in (mindestens) zwei verschiedenen Darstellungen:

1. Im Arbeitsspeicher werden Daten in Objekten, Strukturen, Listen, Arrays, Hashtabellen, Bäumen usw. gespeichert. Diese Datenstrukturen sind für einen effizienten Zugriff und die Manipulation durch die CPU optimiert (typischerweise mit Zeigern).
2. Wenn Sie Daten in eine Datei schreiben oder über das Netzwerk senden wollen, müssen Sie sie in Form einer eigenständigen Bytesequenz codieren (zum Beispiel als JSON-Dokument). Da ein Zeiger für keinen anderen Prozess Sinn machen würde, sieht diese Darstellung als Bytesequenz gänzlich anders aus als die Datenstrukturen, die normalerweise im Arbeitsspeicher verwendet werden.[1]

Wir brauchen also eine Art Übersetzung zwischen den beiden Darstellungen. Die Übersetzung von der speicherinternen Darstellung in eine Bytesequenz heißt *Codierung* (auch als *Serialisierung* oder *Marshalling* bezeichnet) und der entgegengesetzte Vorgang *Decodierung* (*Parsing, Deserialisierung, Unmarshalling*).[2]

**Terminologiekonflikt**

*Serialisierung* wird leider auch im Kontext von Transaktionen verwendet (siehe Kapitel 7), und zwar mit einer vollkommen anderen Bedeutung. Um das Wort nicht überzustrapazieren, bleiben wir in diesem Buch bei *Codierung*, selbst wenn *Serialisierung* der vielleicht gebräuchlichere Begriff ist.

Da dies ein so häufiges Problem ist, stehen unzählige verschiedene Bibliotheken und Codierungsformate zur Auswahl. Die folgenden Abschnitte geben einen kurzen Überblick.

## Sprachspezifische Formate

Viele Programmiersprachen unterstützen von Haus aus die Codierung von speicherinternen Objekten zu Bytesequenzen. Zum Beispiel kennt Java die Schnittstelle `java.io.Serializable` [1], Ruby hat `Marshal` [2], in Python ist es `pickle` [3] usw. Außerdem gibt es viele Bibliotheken von Drittanbietern, etwa Kryo für Java [4].

Diese Codierungsbibliotheken sind sehr komfortabel, weil mit ihnen nur ganz wenig Code erforderlich ist, um speicherinterne Objekte zu speichern und wiederherzustellen. Allerdings gibt es bei ihnen auch eine Anzahl ernsthafter Probleme:

- Die Codierung ist oftmals an eine bestimmte Programmiersprache gebunden, und das Lesen der Daten in einer anderen Sprache ist sehr schwierig. Wenn Sie Daten in einer derartigen Codierung speichern oder übertragen, legen Sie sich für eine möglicherweise sehr lange Zeit auf Ihre aktuelle Programmiersprache fest und schließen die Integration Ihrer Systeme in die anderer Organisationen (die eventuell andere Sprachen verwenden) von vornherein aus.
- Um die Daten in denselben Objekttypen wiederherzustellen, muss die Decodierung in der Lage sein, beliebige Klassen zu instanziieren. Hier liegt eine häufige Quelle für Sicherheitsprobleme [5]: Wenn ein Angreifer Ihre

Anwendung dazu bringen kann, eine beliebige Bytesequenz zu decodieren, kann er beliebige Klassen instanziieren. Das bietet ihm oftmals die Möglichkeit, Schadcode einzuschleusen, um beispielsweise beliebigen Code remote auszuführen [6, 7].

- Die Versionsverwaltung der Daten ist in diesen Bibliotheken oftmals wenig durchdacht: Da sie für schnelle und einfache Codierung der Daten vorgesehen sind, vernachlässigen sie oft die unbequemen Probleme der Vorwärts- und Abwärtskompatibilität.
- Die Effizienz (die benötigte CPU-Zeit zum Codieren oder Decodieren und die Größe der codierten Struktur) spielt oft ebenfalls eine untergeordnete Rolle. Zum Beispiel ist die in Java integrierte Serialisierung berüchtigt für ihre schlechte Performance und aufgeblähte Codierung [8].

Aus diesen Gründen sollten Sie die integrierte Codierung Ihrer Sprache nur für sehr kurzlebige Daten verwenden.

## JSON, XML und binäre Varianten

Wenn man auf standardisierte Codierungen übergeht, die sich durch viele Programmiersprachen schreiben und lesen lassen, sind JSON und XML die Favoriten. Sie sind weithin bekannt, erfahren breite Unterstützung und werden fast genauso wenig gemocht. XML wird oft kritisiert, weil es zu weitschweifig und unnötig kompliziert ist [9]. Die weite Verbreitung von JSON geht hauptsächlich auf die integrierte Unterstützung in Webbrowsern (aufgrund der Tatsache, dass es eine Teilmenge von JavaScript ist) und die Einfachheit relativ zu XML zurück. CSV ist ein weiteres beliebtes sprachunabhängiges Format, auch wenn es weniger leistungsfähig ist.

JSON, XML und CSV sind Textformate und somit für den Menschen einigermaßen verständlich (obwohl die Syntax ein beliebtes Diskussionsthema ist). Abgesehen von den oberflächlichen syntaktischen Problemen weisen sie auch einige subtile Probleme auf:

- Es gibt eine ganze Menge Unklarheiten hinsichtlich der Codierung von Zahlen. In XML und CSV kann man nicht unterscheiden zwischen einer Zahl und einem String, der zufälligerweise aus Ziffern besteht (außer durch Bezugnahme auf ein externes Schema). JSON unterscheidet Strings und Zahlen, macht aber keinen Unterschied zwischen Ganzzahl und Gleitkommazahlen und spezifiziert auch keine Genauigkeit.

Dies ist ein Problem beim Umgang mit großen Zahlen; zum Beispiel lassen sich Ganzzahlen größer als $2^{53}$ in einer Gleitkommazahl doppelter Genauigkeit nach dem Standard IEEE 754 nicht genau darstellen. Solche Zahlen werden also ungenau, wenn sie in einer Sprache geparst werden, die Gleitkommazahlen verwendet (wie zum Beispiel JavaScript). Zahlen größer als $2^{53}$ kommen beispielsweise auf Twitter vor, das jeden Tweet mit einer 64-Bit-Zahl kennzeichnet. Das von der Twitter-API zurückgegebene JSON enthält Tweet-IDs zweimal – einmal als JSON-Zahl und einmal als Dezimalstring, um dem Umstand Rechnung zu tragen, dass die Zahlen sonst von JavaScript-Anwendungen nicht korrekt geparst werden [10].

- JSON und XML sind für Unicode-Zeichenstrings (d.h. Klartext) ausgelegt, unterstützen aber keine binären Strings (Bytesequenzen ohne Zeichencodierung). Binäre Strings sind ein nützliches Feature, und daher umgeht man diese Beschränkung häufig, indem man die Binärdaten per Base64 codiert. Aus dem verwendeten Schema geht dann hervor, dass der Wert als Base64-codiert zu interpretieren ist. Das funktioniert, ist aber etwas umständlich und erhöht die Datenmenge um etwa 33%.
- Es gibt eine optionale Schemaunterstützung sowohl für XML [11] als auch für JSON [12]. Diese Schemasprachen sind recht leistungsfähig und somit auch ziemlich kompliziert zu lernen und zu implementieren. Die Verwendung von XML-Schemas ist durchaus üblich, doch viele JSON-basierte Tools haben mit Schemas nichts am Hut. Da die korrekte Interpretation von Daten (wie zum Beispiel Zahlen und binären Strings) von Informationen im Schema abhängt, müssen Anwendungen, die ohne XML-/JSON-Schemas arbeiten, stattdessen möglicherweise die passende Logik zum Codieren/Decodieren fest im Code realisieren.
- Da CSV keinerlei Schema kennt, liegt es bei der Anwendung, die Bedeutung jeder Zeile und Spalte zu definieren. Wenn eine Anwendung eine neue Zeile oder Spalte hinzufügt, müssen Sie diese Änderung manuell behandeln. Darüber hinaus ist CSV auch ein ziemlich ungenaues Format (was passiert, wenn ein Wert ein Komma oder das Zeichen für eine Zeilenschaltung enthält?) Obwohl Regeln für Escapezeichen formal spezifiziert worden sind [13], implementieren nicht alle Parser diese Regeln korrekt.

Trotz dieser Mängel sind JSON, XML und CSV für viele Zwecke gut geeignet. Es ist absehbar, dass sie weiterhin viel genutzt werden, speziell als Datenaustauschformate (d.h. für das Übermitteln von Daten von einer Organisation an eine andere). Solange sich die Beteiligten über das Format einig

sind, spielt es oftmals keine Rolle, wie ansprechend oder effizient das Format ist. Die Schwierigkeit, verschiedene Organisationen dazu zu bewegen, sich auf irgendein gemeinsames Format zu einigen, wiegt schwerer als die meisten anderen Probleme.

## Binäre Codierung

Bei Daten, die Sie ausschließlich intern in Ihrer Organisation verwenden, ist der Druck geringer, ein Codierungsformat mit dem kleinsten gemeinsamen Nenner zu verwenden. So könnten Sie zum Beispiel ein Format wählen, das kompakter ist oder sich schneller parsen lässt. Für kleine Datenvolumen sind die Gewinne zu vernachlässigen, doch sobald Sie in den Bereich von Terabyte kommen, kann die Wahl des Datenformats einen großen Einfluss haben.

Auch wenn JSON weniger weitschweifig ist als XML, verbrauchen beide Formate trotzdem noch wesentlich mehr Platz als binäre Formate. Diese Beobachtung hat zur Entwicklung einer Fülle von Binärcodierungen für JSON (MessagePack, BSON, BJSON, UBJSON, BISON und Smile, um nur einige zu nennen) und für XML (zum Beispiel WBXML und Fast Infoset) geführt. Diese Formate haben sich in verschiedenen Nischen etablieren können, doch keines von ihnen ist so weit verbreitet wie die Textversionen von JSON und XML.

Einige dieser Formate erweitern den Satz der Datentypen (zum Beispiel unterscheiden sie Ganzzahlen und Gleitkommazahlen oder unterstützen binäre Strings), lassen aber sonst das JSON-/XML-Datenmodell unverändert. Insbesondere schreiben sie kein Schema vor und müssen deshalb sämtliche Objektfeldnamen in die codierten Daten einbinden. In einer binären Codierung des JSON-Dokuments von Beispiel 4-1 müssen also die Strings `userName`, `favoriteNumber` und `interests` enthalten sein.

*Beispiel 4-1: Beispieldatensatz, den wir in diesem Kapitel in mehreren binären Formaten codieren*

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
```

```
}
```

Sehen wir uns ein Beispiel von MessagePack an, einer binären Codierung für JSON. Die in Abbildung 4-1 gezeigte Bytesequenz erhalten Sie, wenn Sie das JSON-Dokument in Beispiel 4-1 mit MessagePack codieren [14]. Die ersten Bytes der Sequenz geben Folgendes an:

1. Das erste Byte, `0×83`, zeigt an, dass die folgenden Daten ein Objekt (obere vier Bits = `0×80`) mit drei Feldern (untere vier Bits = `0×03`) codieren. (Was passiert, wenn ein Objekt mehr als 15 Felder umfasst? Da sich die Anzahl der Felder dann nicht mehr in vier Bits darstellen lässt, erhält das Objekt einen anderen Typindikator, und die Anzahl der Felder wird in zwei oder vier Bytes codiert.)
2. Das zweite Byte, `0×a8`, gibt an, dass die folgenden Bytes einen String (obere vier Bits = `0×a0`) mit einer Länge von acht Bytes (untere vier Bits = `0×08`) codieren.
3. Die nächsten acht Bytes enthalten den Feldnamen `userName` in ASCII. Da die Länge bereits vor dem String steht, ist keine Markierung (oder ein Escapezeichen) notwendig, um das Ende des Strings zu kennzeichnen.
4. Die nächsten sieben Bytes codieren mit dem Präfix `0×a6` und 6 Buchstaben den Stringwert `Martin` usw.

Die binäre Codierung ist 66 Bytes lang und damit nur wenig kürzer als die 81 Bytes, die eine JSON-Textcodierung (mit entfernten Whitespaces) beansprucht. In dieser Hinsicht sind sämtliche Binärcodierungen von JSON einander ähnlich. Es ist nicht klar, ob eine so geringe Platzersparnis und vielleicht ein beschleunigtes Parsing den Verlust der Klartexteigenschaft wettmachen.

Die folgenden Abschnitte zeigen, wie es bedeutend besser geht: Wir codieren denselben Datensatz in nur 32 Bytes.

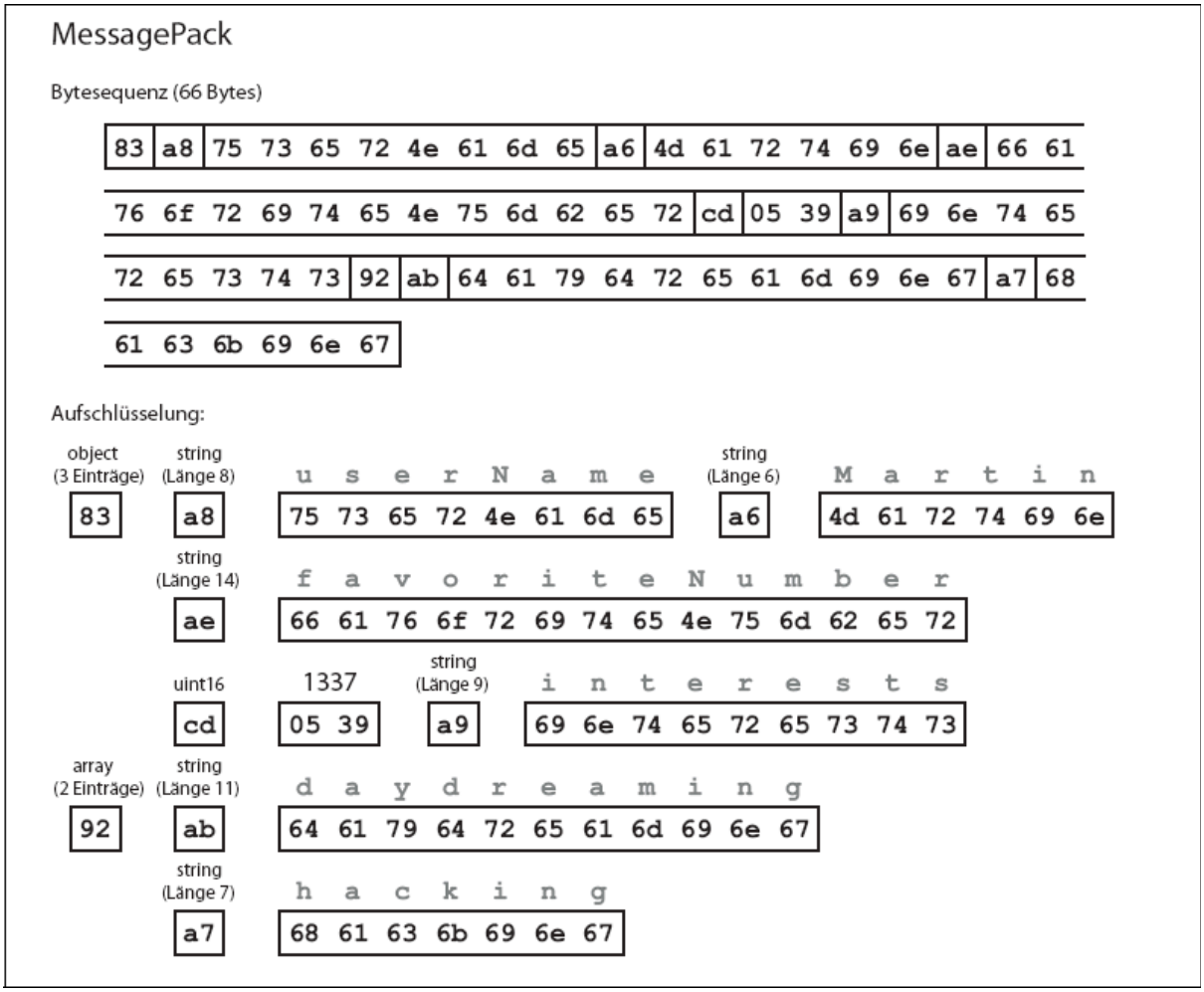



*Abbildung 4-1: Beispieldatensatz (Beispiel 4-1), der mit MessagePack codiert wurde*

## Thrift und Protocol Buffers

Apache Thrift [15] und Protocol Buffers (protobuf) [16] sind Bibliotheken für die binäre Codierung, die auf demselben Prinzip basieren. Protocol Buffers wurde ursprünglich von Google und Thrift von Facebook entwickelt. Beide Projekte sind 2007/2008 zu Open Source gemacht worden [17].

Sowohl Thrift als auch Protocol Buffers setzen ein Schema für die zu codierenden Daten voraus. Um die Daten von Beispiel 4-1 in Thrift zu codieren, beschreiben Sie das Schema in der Thrift-IDL (Interface Description Language), wie es der folgende Code zeigt:

```
struct Person {

  1: required string       userName,
```

```
    2: optional i64          favoriteNumber,
    3: optional list<string> interests
}
```

Die äquivalente Schemadefinition für Protocol Buffers sieht sehr ähnlich aus:

```
message Person {
    required string user_name        = 1;
    optional int64  favorite_number  = 2;
    repeated string interests        = 3;
}
```

Thrift und Protocol Buffers bringen ein Codeerzeugungstool mit, das anhand einer Schemadefinition Klassen erzeugt, die das Schema in verschiedenen Programmiersprachen implementieren [18]. Diesen generierten Code kann Ihre Anwendung aufrufen, um Datensätze des Schemas zu codieren oder zu decodieren.

Wie sehen die Daten aus, die mit diesem Schema codiert wurden? Verwirrenderweise hat Thrift mit BinaryProtocol und CompactProtocol zwei verschiedene Formate zur Binärcodierung.[3] Sehen wir uns zuerst BinaryProtocol an. Codiert man Beispiel 4-1 in diesem Format, ergeben sich 59 Bytes, wie sie in Abbildung 4-2 dargestellt sind [19].

Ähnlich wie in Abbildung 4-1 hat jedes Feld eine Typanmerkung (um anzuzeigen, ob es sich um einen String, eine Ganzzahl, eine Liste usw. handelt) und bei Bedarf ein Längenkennzeichen (Länge eines Strings, Anzahl der Elemente in einer Liste). Die Strings, die in den Daten erscheinen (`"Martin"`, `"daydreaming"`, `"hacking"`) sind ähnlich wie zuvor als ASCII (oder vielmehr UTF-8) codiert.

Der große Unterschied gegenüber Abbildung 4-1 besteht darin, dass es keine Feldnamen (`userName`, `favoriteNumber`, `interests`) gibt. Stattdessen enthalten die codierten Daten *Feldtags*, und zwar in Form von Zahlen (1, 2 und 3). Dabei handelt es sich um die Zahlen, die in der Schemadefinition erscheinen. Feldtags sind so etwas wie Alias für Felder – damit lässt sich in kompakter Form sagen, über welches Feld wir sprechen, ohne den Feldnamen ausschreiben zu müssen.

Die CompactProtocol-Codierung von Thrift ist semantisch dem BinaryProtocol äquivalent, doch wie Abbildung 4-3 zeigt, bringt das Format dieselben Informationen in nur 34 Bytes unter. Dazu packt es den Feldtyp und die Tagnummer in einem einzelnen Byte zusammen und verwendet Ganzzahlen variabler Länge. Anstatt volle acht Bytes für die Zahl 1337 zu verbrauchen, kommt das Format mit zwei Bytes aus. Dabei zeigt das höchste Bit jedes Bytes an, ob noch weitere Bytes folgen. Folglich werden Zahlen zwischen –64 und 63 in einem Byte codiert, Zahlen zwischen –8192 und 8191 in zwei Bytes usw. Größere Zahlen benötigen mehr Bytes.

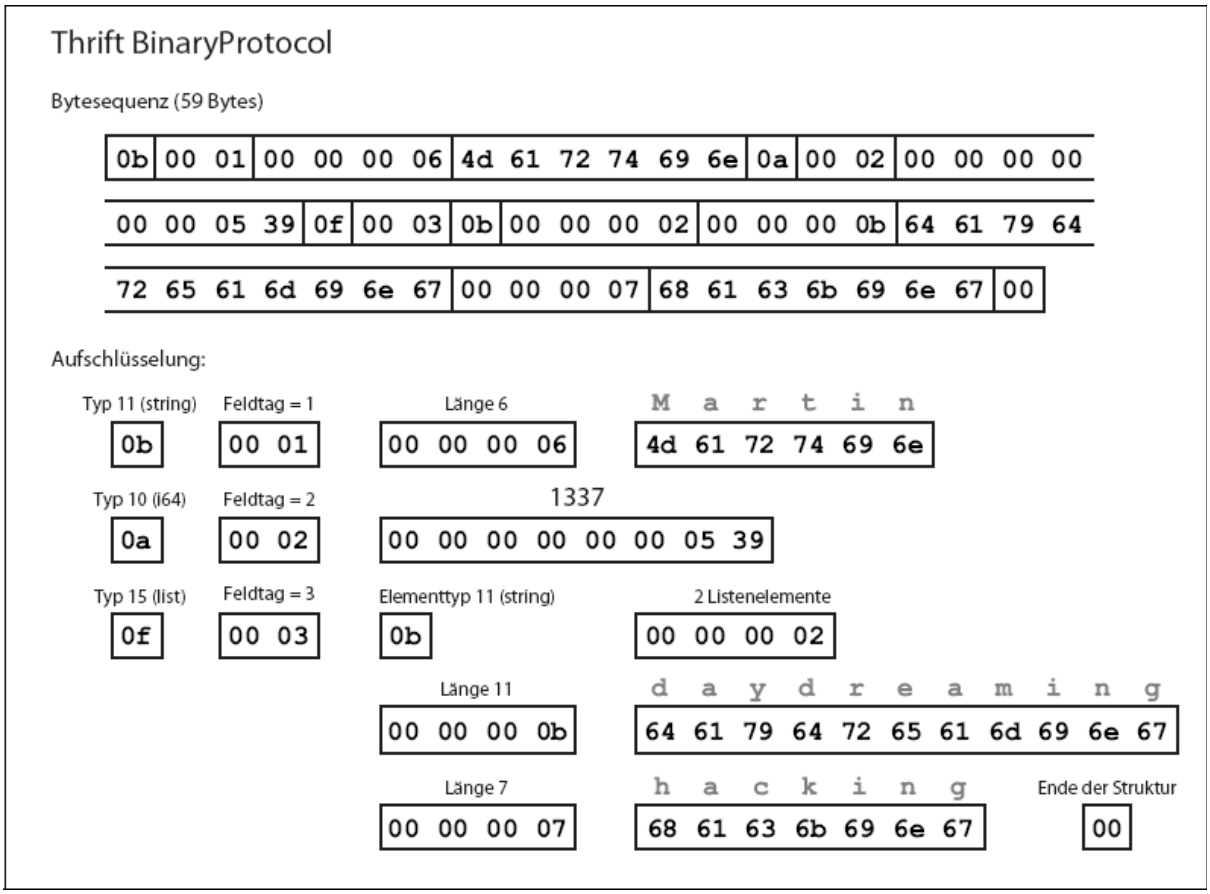



*Abbildung 4-2: Beispieldatensatz, codiert mit BinaryProtocol von Thrift*

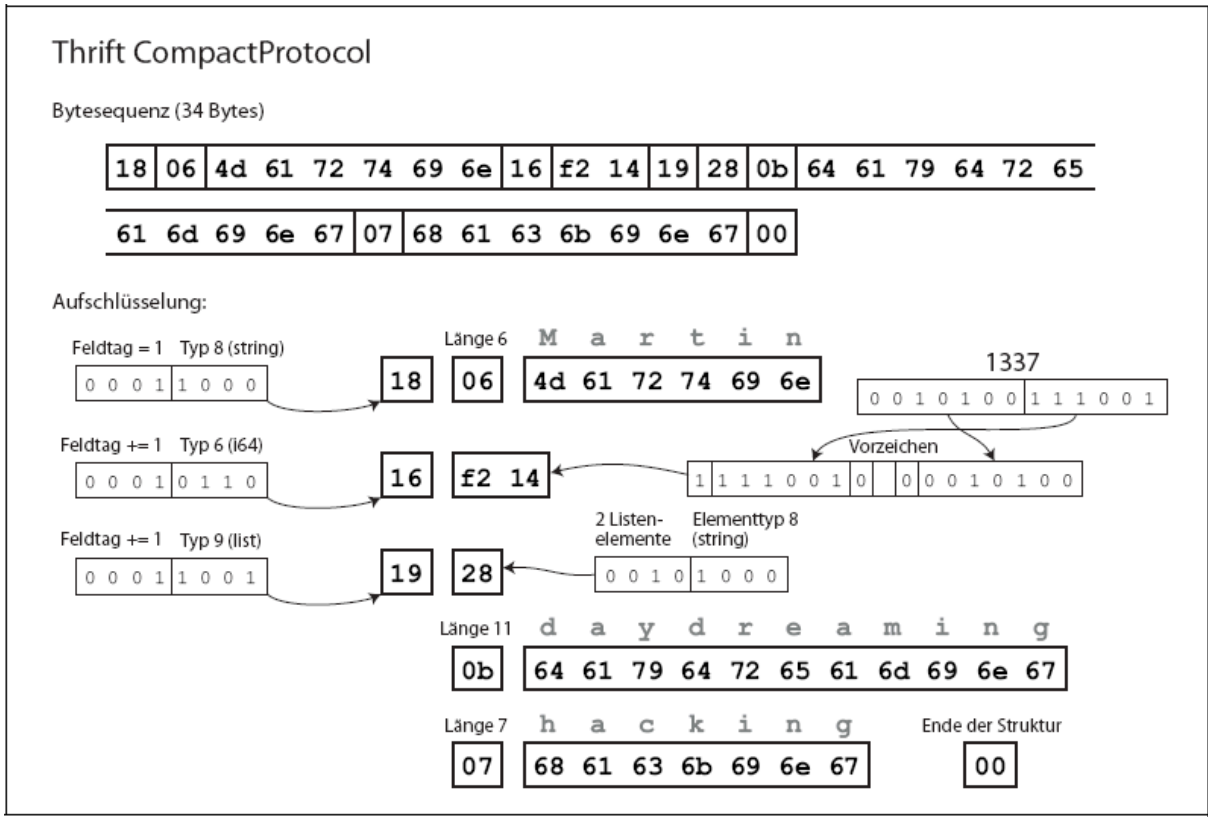



*Abbildung 4-3: Beispieldatensatz, der mit CompactProtocol von Thrift codiert ist*

Schließlich codiert Protocol Buffers (das nur ein binäres Format kennt) dieselben Daten wie in Abbildung 4-4 dargestellt. Die Bitpackung sieht ein wenig anders aus, doch sonst ist das Format dem CompactProtocol von Thrift sehr ähnlich. Bei Protocol Buffers passt derselbe Datensatz in 33 Bytes.

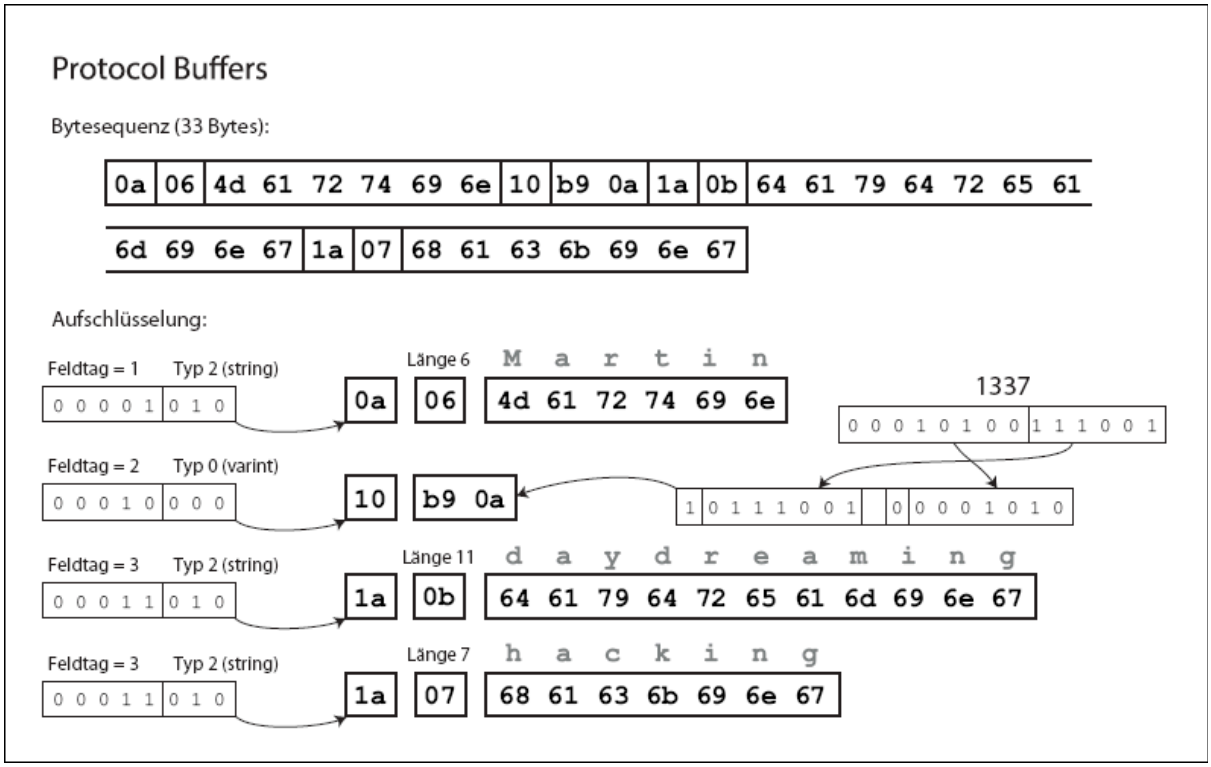



*Abbildung 4-4: Beispieldatensatz, der mit Protocol Buffers codiert ist*

Ein Detail ist noch anzumerken: In den weiter vorn gezeigten Schemas war jedes Feld entweder als `required` (erforderlich) oder `optional` markiert. Für die Codierung des Felds ist das aber nicht relevant (nichts in den binären Daten zeigt an, ob ein Feld erforderlich war). Der Unterschied ist einfach der, dass `required` eine Laufzeitprüfung erlaubt, die scheitert, wenn das Feld nicht gesetzt ist. Dies kann für die Fehlersuche nützlich sein.

## Feldtags und Schemaevolution

Wie weiter oben erläutert, müssen sich Schemas im Lauf der Zeit zwangsläufig ändern. Wir nennen das *Schemaevolution*. Was machen Thrift und Protocol Buffers mit Schemaänderungen, um Abwärts- und Vorwärtskompatibilität zu bewahren?

Wie Sie den Beispielen entnehmen können, ist ein codierter Datensatz lediglich die Verkettung seiner codierten Felder. Jedes Feld wird durch seine Tagnummer gekennzeichnet (die Zahlen 1, 2, 3 in den Beispielschemas) und mit einem Datentyp (zum Beispiel String oder Integer) annotiert. Wenn ein Feldwert nicht festgelegt ist, wird er im codierten Datensatz einfach weggelassen. Hieraus können Sie ersehen, dass Feldtags für die Bedeutung der codierten Daten entscheidend sind. Den Namen eines Felds können Sie im Schema ändern, da

die codierten Daten niemals auf die Feldnamen verweisen. Dagegen dürfen Sie das Tag eines Felds nicht ändern, denn das würde alle vorhandenen codierten Daten ungültig machen.

Dem Schema können Sie neue Felder hinzufügen, sofern Sie jedem Feld eine neue Tagnummer zuweisen. Versucht nun alter Code (der von den hinzugefügten neuen Tagnummern nichts weiß) die von neuem Code geschriebenen Daten zu lesen – einschließlich eines neuen Felds mit einer Tagnummer, die er nicht erkennt –, kann er dieses Feld einfach ignorieren. Anhand der Datentypannotation kann der Parser ermitteln, wie viele Bytes er überspringen muss. Dadurch bleibt Vorwärtskompatibilität gewahrt: Alter Code kann also Datensätze lesen, die von neuem Code geschrieben wurden.

Wie sieht es mit Abwärtskompatibilität aus? Solange jedes Feld eine eindeutige Tagnummer hat, kann neuer Code immer alte Daten lesen, weil die Tagnummern immer noch dieselbe Bedeutung haben. Die einzige Einschränkung ist, dass Sie ein neu hinzugefügtes Feld nicht zu einem erforderlichen Feld machen können. Würden Sie ein Feld hinzufügen und es als erforderlich festlegen, würde diese Überprüfung scheitern, wenn neuer Code Daten liest, die mit altem Code geschrieben wurden, weil der alte Code das von Ihnen neu hinzugefügte Feld nicht geschrieben hat. Um also Abwärtskompatibilität zu wahren, muss jedes Feld, das Sie nach der anfänglichen Bereitstellung des Schemas hinzufügen, optional sein oder einen Standardwert haben.

Ein Feld zu entfernen, ist wie das Hinzufügen eines Felds, wobei sich die Fragen der Abwärts- und Vorwärtskompatibilität umkehren. Das heißt, Sie können nur ein optionales Feld entfernen (ein erforderliches Feld kann nie entfernt werden) und Sie dürfen nie dieselbe Tagnummer erneut verwenden (weil es irgendwo noch geschriebene Daten geben kann, die die alte Tagnummer enthalten, und dieses Feld vom neuen Code ignoriert werden muss).

## Datentypen und Schemaevolution

Ist es möglich, den Datentyp eines Felds zu ändern? Das kann zulässig sein (informieren Sie sich über Details in der Dokumentation), doch es besteht die Gefahr, dass Werte an Genauigkeit verlieren oder abgeschnitten werden. Nehmen wir zum Beispiel an, Sie ändern eine 32-Bit-Ganzzahl in eine 64-Bit-Ganzzahl. Neuer Code liest problemlos die Daten, die der alte Code geschrieben hat, weil der Parser die fehlenden Bits mit Nullen auffüllen kann. Wenn jedoch

alter Code Daten liest, die neuer Code geschrieben hat, verwendet der alte Code trotzdem eine 32-Bit-Variable, um den Wert zu speichern. Wenn der decodierte 64-Bit-Wert nicht in 32 Bits passt, wird er abgeschnitten.

Protocol Buffers hat die Eigenheit, dass es keinen Listen- oder Array-Datentyp kennt. Stattdessen verwendet es eine `repeated`-Markierung für Felder (eine dritte Option neben `required` und `optional`). Wie Abbildung 4-4 zeigt, hält die Codierung eines wiederholten (`repeated`) Felds genau das, was sie verspricht: Das gleiche Feldtag erscheint einfach mehrmals im Datensatz. Dies hat den angenehmen Effekt, dass sich ein optionales (einwertiges) Feld zu einem wiederholten (mehrwertigen) Feld umwandeln lässt. Neuer Code, der alte Daten liest, sieht eine Liste mit null oder einem Element vor sich (je nachdem, ob das Feld vorhanden war), während alter Code, der neue Daten liest, nur das letzte Element der Liste sieht.

Bei Thrift gibt es einen dedizierten Listendatentyp, der mit dem Datentyp der Listenelemente parametrisiert wird. Dies ermöglicht zwar nicht die gleiche Evolution von einwertigen zu mehrwertigen Feldern wie bei Protocol Buffers, bietet dafür aber den Vorteil, verschachtelte Listen zu unterstützen.

## Avro

Apache Avro [20] ist ein anderes Codierungsformat, das sich in interessanten Punkten von Protocol Buffers und Thrift unterscheidet. Es wurde 2009 als Teilprojekt von Hadoop gestartet, da sich Thrift für die Einsatzfälle von Hadoop als nicht geeignet erwiesen hatte [21].

Auch Avro verwendet ein Schema, um die Struktur der zu codierenden Daten zu spezifizieren. Es verfügt über zwei Schemasprachen: Die eine (Avro IDL) ist als Bearbeitungssprache für den Menschen vorgesehen, und die andere (die auf JSON basiert) ist als besser maschinenlesbare Sprache konzipiert.

Unser Beispielschema könnte in Avro IDL wie folgt aussehen:

```
record Person {
    string              userName;
    union { null, long } favoriteNumber = null;
```

```
        array<string>        interests;

    }
```

Die äquivalente JSON-Darstellung dieses Schemas lautet:

```
    {

        "type": "record",

        "name": "Person",

        "fields": [

            {"name": "userName",       "type": "string"},

            {"name": "favoriteNumber", "type": ["null",
            "long"], "default": null},

            {"name": "interests",      "type": {"type":
            "array", "items": "string"}}

        ]

    }
```

Zunächst einmal ist festzustellen, dass es im Schema keine Tagnummern gibt. Wenn wir unseren Beispieldatensatz (Beispiel 4-1) mit diesem Schema codieren, ist die Binärcodierung mit Avro lediglich 32 Bytes lang – die kompakteste aller Codierungen, die Sie bisher kennengelernt haben. Die codierte Bytesequenz ist in Abbildung 4-5 aufgegliedert dargestellt.

Wie eine Analyse der Bytesequenz zeigt, gibt es nichts, um Felder oder deren Datentypen zu identifizieren. Die Codierung besteht einfach aus Werten, die miteinander verkettet sind. Ein String besteht aus einem Längenpräfix, gefolgt von UTF-8-Bytes, doch aus den codierten Daten geht nicht hervor, dass es sich

um einen String handelt. Es könnte genauso gut eine Ganzzahl oder etwas vollkommen anderes sein. Eine Ganzzahl wird mit einer variablen Länge codiert (genau wie beim CompactProtocol von Thrift).

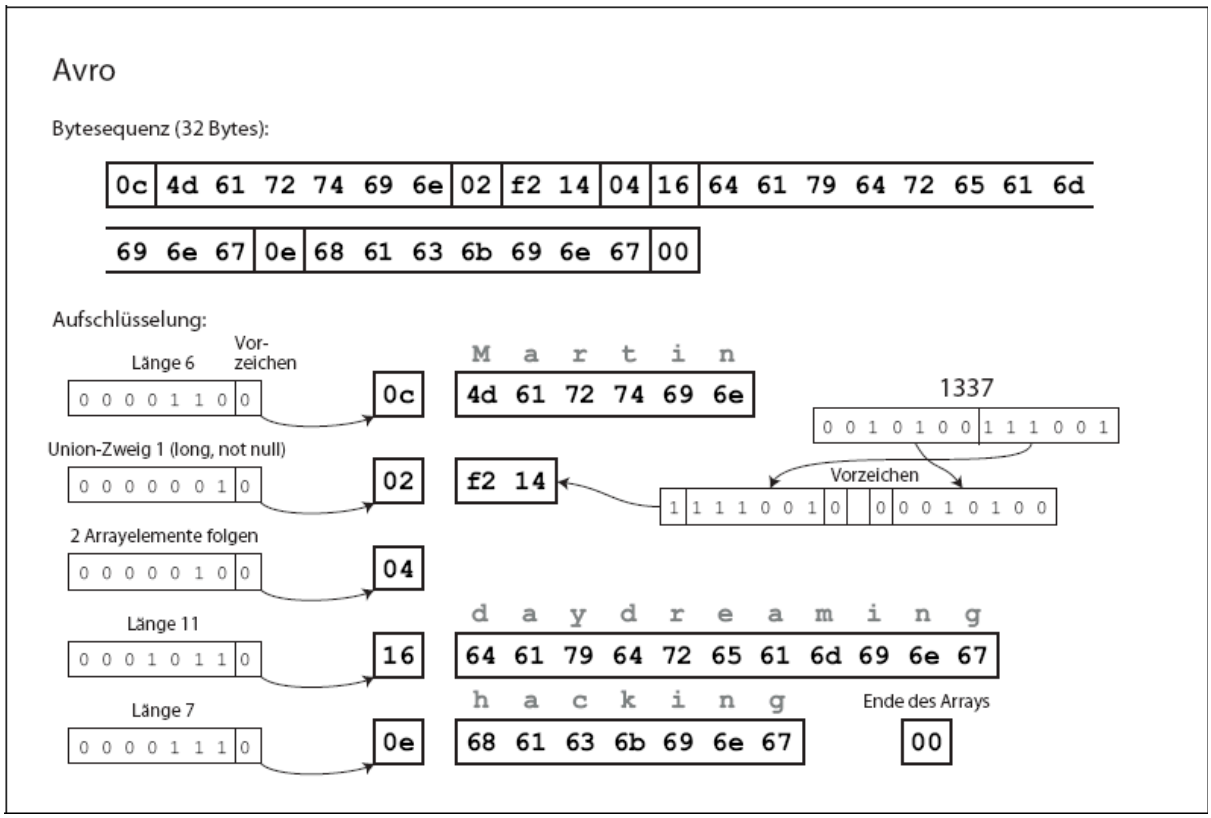



*Abbildung 4-5: Beispieldatensatz, der mit Avro codiert wurde*

Um die Binärdaten zu parsen, gehen Sie durch die Felder in der Reihenfolge, in der sie im Schema erscheinen, und entnehmen dem Schema den Datentyp für jedes Feld. Folglich lassen sich die Binärdaten nur dann korrekt decodieren, wenn der Code, der die Daten liest, *genau das gleiche Schema* verwendet wie der Code, der die Daten geschrieben hat. Jede Abweichung im Schema zwischen dem Leser und dem Schreiber würde falsch decodierte Daten bedeuten.

Wie unterstützt dann Avro eine Schemaevolution?

## Das Schema des Schreibers und das Schema des Lesers

Wenn eine Anwendung bestimmte Daten mit Avro codieren möchte (in eine Datei oder Datenbank schreiben, über das Netzwerk senden usw.), codiert sie die Daten mit der Schemaversion, die ihr bekannt ist – zum Beispiel kann das Schema fester Bestandteil des Anwendungscodes sein. Dies wird als *Schema des Schreibers* bezeichnet.

Möchte eine Anwendung bestimmte Daten decodieren (aus einer Datei oder Datenbank lesen, über das Netzwerk empfangen usw.), geht sie davon aus, dass die Daten in einem bestimmten Schema vorliegen, dem sogenannten *Schema des Lesers*. Das ist das Schema, auf das sich der Anwendungscode stützt – von diesem Schema kann der Code beim Erstellen der Anwendung generiert worden sein.

Der Schlüsselgedanke bei Avro ist, dass das Schema des Schreibers und das Schema des Lesers *nicht gleich sein müssen* – sie müssen nur kompatibel sein. Wenn Daten decodiert (gelesen) werden, löst die Avro-Bibliothek die Unterschiede auf, indem sie das Schema des Schreibers und das Schema des Lesers nebeneinander betrachtet und die Daten vom Schema des Schreibers in das Schema des Lesers übersetzt. Die Avro-Spezifikation [20] definiert genau, wie diese Auflösung funktioniert. Abbildung 4-6 veranschaulicht die Abläufe.

Zum Beispiel ist es kein Problem, wenn im Schema des Schreibers und im Schema des Lesers die Felder in einer anderen Reihenfolge erscheinen, weil die Schemaauflösung die Felder nach dem Namen abgleicht. Trifft der Code, der die Daten liest, auf ein Feld, das im Schema des Schreibers, aber nicht im Schema des Lesers erscheint, ignoriert er es. Wenn der Daten lesende Code ein bestimmtes Feld erwartet, das Schema des Schreibers aber kein Feld dieses Namens enthält, wird es mit einem Standardwert gefüllt, der im Schema des Lesers deklariert ist.

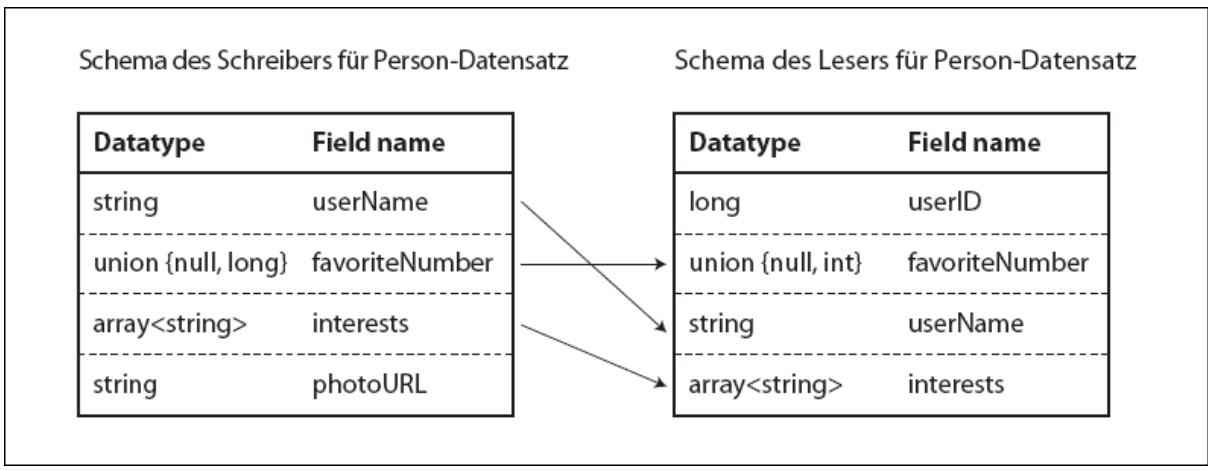



*Abbildung 4-6: Ein Avro-Leser löst Unterschiede zwischen dem Schema des Schreibers und dem Schema des Lesers auf.*

## Regeln für Schemaevolution

Bei Avro bedeutet Vorwärtskompatibilität, dass Sie eine neue Version des Schemas als Schreiber und eine alte Version des Schemas als Leser haben können. Umgekehrt heißt Abwärtskompatibilität, dass Sie eine neue Version des Schemas als Leser und eine alte Version als Schreiber verarbeiten können.

Um die Kompatibilität zu wahren, sollten Sie nur Felder, die Standardwerte besitzen, hinzufügen oder entfernen. (Das Feld `favoriteNumber` in unserem Avro-Schema hat den Standardwert `null`.) Nehmen wir als Beispiel an, Sie fügen ein Feld mit einem Standardwert hinzu, sodass dieses neue Feld im neuen Schema existiert, aber nicht im alten. Wenn ein Leser, der das neue Schema verwendet, einen Datensatz liest, der mit dem alten Schema geschrieben wurde, setzt er den Standardwert für das fehlende Feld ein.

Würden Sie ein Feld hinzufügen, das keinen Standardwert besitzt, wären neue Leser nicht in der Lage, die von alten Schreibern geschriebenen Daten zu lesen. Damit würden Sie die Abwärtskompatibilität verletzen. Wenn Sie ein Feld entfernen würden, das keinen Standardwert hat, wären alte Leser nicht in der Lage, die von neuen Schreibern geschriebenen Daten zu lesen. Hiermit würden Sie also die Vorwärtskompatibilität verletzen.

In manchen Programmiersprachen ist `null` ein akzeptabler Standardwert für jede Variable, in Avro jedoch nicht: Wenn Sie den Wert `null` für ein Feld zulassen möchten, müssen Sie einen `union`-Typ bemühen. Zum Beispiel zeigt `union { null, long, string } field;` an, dass `field` eine Zahl oder ein String oder `null` sein kann. Den Wert `null` können Sie nur dann als Standardwert verwenden, wenn er einer der Union-Zweige ist.[4] Dies ist etwas umständlicher als für alle Variablen standardmäßig `null` zuzulassen, verringert aber die Gefahr von Fehlern, weil explizit ausgedrückt wird, was `null` sein darf und was nicht [22].

Somit kennt Avro auch keine Markierungen wie `optional` und `required`, die bei Protocol Buffers und Thrift vorhanden sind (dafür gibt es `union`-Typen und Standardwerte). Es ist möglich, den Datentyp eines Felds zu ändern, sofern Avro den Typ konvertieren kann. Auch der Name eines Felds lässt sich ändern, doch ist das etwas diffizil: Das Schema des Lesers kann Alias für Feldnamen enthalten, sodass es die Schemafeldnamen eines alten Schreibers mit den Alias abgleichen kann. Das Ändern eines Feldnamens ist also abwärtskompatibel, aber nicht vorwärtskompatibel. Analog dazu ist das Hinzufügen eines Zweigs zu einem Union-Typ abwärtskompatibel, aber nicht vorwärtskompatibel.

## Doch was ist das Schema des Schreibers?

Bisher haben wir eine wichtige Frage vernachlässigt: Woher kennt der Leser das Schema des Schreibers, mit dem ein bestimmter Abschnitt der Daten codiert wurde? Wir können nicht einfach das gesamte Schema in jeden Datensatz einschließen, weil das Schema wahrscheinlich viel größer sein würde als die codierten Daten. Dies würde die Platzeinsparung durch die binäre Codierung ad absurdum führen.

Die Antwort hängt vom Kontext ab, in dem Avro verwendet wird. Hierzu einige Beispiele:

*Große Datei mit vielen Datensätzen*

> Avro wird häufig – insbesondere im Kontext von Hadoop – für das Speichern einer großen Datei mit Millionen von Datensätzen eingesetzt, die alle mit demselben Schema codiert sind. (Auf derartige Situationen geht Kapitel 10 ein.) In diesem Fall kann der Schreiber dieser Datei einfach das Schema des Schreibers einmal am Anfang der Datei einbinden. Avro spezifiziert hierfür ein Dateiformat (Objektcontainerdateien).

*Datenbank mit individuell geschriebenen Datensätzen*

> In einer Datenbank können verschiedene Datensätze zu verschiedenen Zeitpunkten mit verschiedenen Schemas des Schreibers geschrieben werden – man kann nicht davon ausgehen, dass alle Datensätze nach demselben Schema aufgebaut sind. Am einfachsten ist es, jeden Datensatz am Anfang mit einer Versionsnummer zu versehen und eine Liste der Schemaversionen in der Datenbank zu führen. Ein Leser kann einen Datensatz abrufen, die Versionsnummer herausziehen und dann das Schema des Schreibers für diese Versionsnummer aus der Datenbank holen. Anhand dieses Schemas des Schreibers lässt sich dann der Rest des Datensatzes decodieren. (Zum Beispiel funktioniert Espresso [23] auf diese Weise.)

*Datensätze über eine Netzwerkverbindung senden*

> Wenn zwei Prozesse über eine bidirektionale Netzwerkverbindung kommunizieren, können sie die Schemaversion beim Verbindungsaufbau aushandeln und dann dieses Schema für die Dauer der Verbindung verwenden. So arbeitet auch das RPC-Protokoll von Avro (siehe Abschnitt »Datenfluss über Dienste: REST und RPC« auf Seite 140).

Eine Datenbank von Schemaversionen ist auf jeden Fall nützlich, da sie als Dokumentation dient und Ihnen erlaubt, die Schemakompatibilität zu überprüfen [24]. Als Versionsnummer können Sie eine einfache fortlaufende Ganzzahl oder einen Hashwert des Schemas verwenden.

## Dynamisch generierte Schemas

Verglichen mit Protocol Buffers und Thrift hat Avro unter anderem den Vorteil, dass im Schema keine Tagnummern enthalten sind. Doch warum ist das wichtig? Wo liegt das Problem, eine Reihe von Zahlen im Schema mitzuführen?

Der Unterschied ist, dass Avro *dynamisch generierten* Schemas entgegenkommt. Nehmen wir zum Beispiel an, Sie möchten den Inhalt einer relationalen Datenbank in eine Datei ausgeben, und zwar in einem Binärformat, um die oben erwähnten Probleme mit Textformaten (JSON, CSV, SQL) zu vermeiden. Mit Avro können Sie ziemlich einfach ein Avro-Schema (in der weiter oben vorgestellten JSON-Darstellung) aus dem relationalen Schema erzeugen, den Datenbankinhalt mit diesem Schema codieren und somit alles in eine Avro-Objektcontainerdatei ausgeben [25]. Für jede Datenbanktabelle generieren Sie ein Datensatzschema, und jede Spalte wird zu einem Feld in diesem Datensatz. Der Spaltenname in der Datenbank wird auf den Feldnamen in Avro abgebildet.

Ändert sich nun das Datenbankschema (weil zum Beispiel einer Tabelle eine Spalte hinzugefügt und eine Spalte entfernt wurde), können Sie einfach ein neues Avro-Schema aus dem aktualisierten Datenbankschema erzeugen und Daten im neuen Avro-Schema exportieren. Der Exportvorgang braucht sich um die Schemaänderung nicht zu kümmern – jedes Mal, wenn er ausgeführt wird, kann er einfach die Schemaumwandlung vornehmen. Jeder, der die neuen Datendateien liest, sieht, dass sich die Felder des Datensatzes geändert haben. Doch da die Felder durch den Namen identifiziert werden, kann das aktualisierte Schema des Schreibers trotzdem noch mit dem Schema des alten Lesers abgeglichen werden.

Wenn Sie dagegen Thrift oder Protocol Buffers für diesen Zweck heranziehen, müssten Sie die Feldtags wahrscheinlich manuell zuweisen: Jedes Mal, wenn sich das Datenbankschema ändert, müsste ein Administrator die Zuordnung der Datenbankspaltennamen zu den Feldtags manuell aktualisieren. (Dies ließe sich vielleicht automatisieren, doch der Schemagenerator müsste sehr genau darauf aufpassen, keine der früher verwendeten Feldtags zuzuweisen.) Diese Art von

dynamisch generiertem Schema war einfach kein Entwurfsziel von Thrift oder Protocol Buffers, bei Avro aber schon.

## Codeerzeugung und dynamisch typisierte Sprachen

Thrift und Protocol Buffers verlassen sich auf Codegenerierung: Nachdem ein Schema definiert worden ist, können Sie Code generieren, der dieses Schema in einer Programmiersprache Ihrer Wahl implementiert. Dies ist in statisch typisierten Sprachen wie Java, C++ oder C# nützlich, weil sich dann effiziente speicherinterne Strukturen für decodierte Daten verwenden lassen und eine Typprüfung und Autovervollständigung in IDEs möglich ist, wenn man Programme schreibt, die auf die Datenstrukturen zugreifen.

In dynamisch typisierten Programmiersprachen wie JavaScript, Ruby oder Python ergibt es wenig Sinn, Code zu generieren, weil es dort normalerweise keine Kompilierung mit Typprüfung gibt. Die Codegenerierung ist in diesen Sprachen oftmals verpönt, da dann ein zusätzlicher Kompilierungsschritt erforderlich wäre. Darüber hinaus ist im Falle eines dynamisch generierten Schemas (wie zum Beispiel eines Avro-Schemas, das aus einer Datenbanktabelle generiert wurde) die Codegenerierung ein unnötiges Hindernis, um an die Daten zu gelangen.

Avro bietet optional eine Codegenerierung für statisch typisierte Programmiersprachen, doch es kann genauso gut auch ohne Generierung verwendet werden. Wenn Sie eine Objektcontainerdatei haben (die das Schema des Schreibers einbettet), können Sie sie einfach mithilfe der Avro-Bibliothek öffnen und sich die Daten in der gleichen Weise wie eine JSON-Datei ansehen. Die Datei ist *selbstbeschreibend*, da sie alle notwendigen Metadaten enthält.

Diese Eigenschaft ist vor allem nützlich in Verbindung mit dynamisch typisierten Datenverarbeitungssprachen wie Apache Pig [26]. In Pig können Sie einfach einige Avro-Dateien öffnen, sie analysieren und abgeleitete Datensätze in Ausgabedateien im Avro-Format schreiben, ohne sich überhaupt über die Schemas Gedanken zu machen.

## Die Vorzüge von Schemas

Wie wir gesehen haben, verwenden Protocol Buffers, Thrift und Avro ein Schema, um ein binäres Codierungsformat zu beschreiben. Ihre Schemasprachen sind wesentlich einfacher als XML Schema oder JSON Schema, die viel detailliertere Validierungsregeln unterstützen (zum Beispiel »der

Stringwert dieses Felds muss diesem regulären Ausdruck entsprechen« oder »der Ganzzahlwert dieses Felds muss zwischen 0 und 100 liegen«) Da Protocol Buffers, Thrift und Avro einfacher zu implementieren und zu verwenden sind, unterstützen sie eine zunehmend breitere Palette von Programmiersprachen.

Diese Codierungen basieren auf Ideen, die keineswegs neu sind. Zum Beispiel haben sie viel gemein mit ASN.1, einer Schemadefinitionssprache, die erstmals 1984 standardisiert wurde [27]. Damit hat man verschiedene Netzwerkprotokolle definiert, und ihre Binärcodierung (DER) ist immer noch in Gebrauch, um beispielsweise SSL-Zertifikate (X.509) zu codieren [28]. ASN.1 unterstützt Schemaevolution mithilfe von Tagnummern, ähnlich wie Protocol Buffers und Thrift [29]. Allerdings ist sie auch sehr komplex und schlecht dokumentiert, sodass ASN.1 wahrscheinlich keine gute Wahl für neue Anwendungen ist.

Viele Datensysteme implementieren auch eine Art proprietärer Binärcodierung für ihre Daten. So haben die meisten relationalen Datenbanken ein Netzwerkprotokoll, über das Sie Abfragen an die Datenbank senden und Antworten zurückbekommen können. Diese Protokolle sind im Allgemeinen für eine bestimmte Datenbank spezifisch, und der Datenbankanbieter stellt einen Treiber bereit (zum Beispiel mit den APIs ODBC oder JDBC), der Antworten aus dem Netzwerkprotokoll der Datenbank in speicherinterne Datenstrukturen decodiert.

Wir stellen also fest, dass Textdatenformate wie JSON, XML und CSV zwar weitverbreitet sind, aber auch Binärcodierungen auf der Basis von Schemas eine durchaus sinnvolle Option sind. Zudem besitzen sie eine Reihe attraktiver Eigenschaften:

- Sie sind möglicherweise wesentlich kompakter als die verschiedenen »binäres JSON«-Varianten, da die codierten Daten keine Feldnamen enthalten müssen.
- Das Schema ist eine wertvolle Form der Dokumentation, und weil es für die Decodierung erforderlich ist, können Sie sicher sein, dass es aktuell ist (während eine manuell gepflegte Dokumentation leicht von der Realität abweichen kann).
- Mit einer Datenbank von Schemas haben Sie die Möglichkeit, die Vorwärts- und Abwärtskompatibilität von Schemaänderungen zu überprüfen, bevor die Änderung bekanntgegeben wird.

- Für Benutzer von statisch typisierten Programmiersprachen ist die Möglichkeit, Code aus dem Schema zu erzeugen, sinnvoll, da damit eine Typprüfung zur Kompilierzeit durchführbar ist.

Alles in allem ermöglicht die Schemaevolution die gleiche Art von Flexibilität wie schemalose/Schema-on-read-JSON-Datenbanken (siehe Abschnitt »Schemaflexibilität im Dokumentmodell« auf Seite 42) und bietet dabei gleichzeitig bessere Garantien für Ihre Daten und bessere Werkzeuge.

# Datenflussmodi

Zu Beginn dieses Kapitel haben wir festgestellt, dass Sie Daten immer dann als Bytesequenz codieren müssen, wenn Sie sie an einen anderen Prozess senden möchten, mit dem Sie keinen Speicher teilen – beispielsweise wenn Sie Daten über das Netzwerk verschicken oder in eine Datei schreiben wollen. Wir haben dann eine breite Palette verschiedener Codierungen für diese Aufgabe besprochen.

Dabei ging es auch um Vorwärts- und Abwärtskompatibilität, die wichtig sind für Evolvierbarkeit (Änderungen erleichtern, indem Sie verschiedene Teile Ihres Systems unabhängig voneinander aktualisieren dürfen und nicht alles auf einmal ändern müssen). Kompatibilität ist eine Beziehung zwischen dem einen Prozess, der Daten codiert, und einem anderen Prozesse, der sie decodiert.

Das ist eine ziemlich abstrakte Idee – es gibt verschiedene Wege, auf denen Daten von einem Prozess zu einem anderen fließen können. Wer codiert die Daten und wer decodiert sie? Im Rest dieses Kapitels untersuchen wir die gängigsten Methoden, wie Daten zwischen Prozessen fließen:

- Über Datenbanken (siehe Abschnitt »Datenfluss über Datenbanken« auf Seite 137)
- Über Dienstaufrufe (siehe Abschnitt »Datenfluss über Dienste: REST und RPC« auf Seite 140)
- Über asynchronen Nachrichtenaustausch (siehe Abschnitt »Datenfluss beim Nachrichtenaustausch« auf Seite 146)

## Datenfluss über Datenbanken

Der Prozess, der in die Datenbank schreibt, codiert die Daten, und der Prozess, der aus der Datenbank liest, decodiert sie. Wenn lediglich ein einzelner Prozess auf die Datenbank zugreift, ist der Leser einfach eine spätere Version desselben

Prozesses – etwas in der Datenbank zu speichern, können Sie sich dann *wie eine Nachricht vorstellen, die Sie an Ihr zukünftiges Ich senden*.

Hier ist zweifellos Abwärtskompatibilität erforderlich, da andernfalls Ihr zukünftiges Ich nicht in der Lage wäre, die Nachricht zu decodieren, die Sie vorher geschrieben haben.

Im Allgemeinen greifen mehrere verschiedene Prozesse zur gleichen Zeit auf eine Datenbank zu. Diese Prozesse können mehrere verschiedene Anwendungen oder Dienste sein oder auch einfach mehrere Instanzen desselben Diensts (die aus Gründen der Skalierbarkeit oder Fehlertoleranz parallel laufen). In jedem Fall ist es in einer Umgebung, in der sich die Anwendung ändert, wahrscheinlich, dass einige Prozesse, die auf die Datenbank zugreifen, neueren Code ausführen, und einige älteren Code – zum Beispiel weil eine neue Version gerade in einem Rolling Upgrade bereitgestellt wird, sodass manche Instanzen bereits aktualisiert wurden, andere aber noch nicht.

Das heißt, dass ein Wert in der Datenbank von einer *neueren* Codeversion geschrieben und später von einer *älteren* Codeversion, die immer noch läuft, gelesen werden kann. Deshalb sind Datenbanken oftmals auch auf Vorwärtskompatibilität angewiesen.

Allerdings gibt es einen zusätzlichen Haken. Angenommen, Sie fügen einem Datensatzschema ein Feld hinzu und der neuere Code schreibt einen Wert für dieses neue Feld in die Datenbank. Später liest eine ältere Codeversion (die das neue Feld nicht kennt) den Datensatz, aktualisiert ihn und schreibt ihn zurück. In dieser Situation sollte der alte Code das neue Feld unversehrt lassen, selbst wenn es sich nicht interpretieren lässt.

Die weiter oben besprochenen Codierungsformate unterstützen eine solche Bewahrung unbekannter Felder, doch manchmal müssen Sie auf Anwendungsebene vorsichtig sein, wie Abbildung 4-7 veranschaulicht. Wenn Sie zum Beispiel in der Anwendung einen Datenbankwert in Modellobjekte decodieren und später diese Modellobjekte erneut codieren, kann das unbekannte Feld in diesem Übersetzungsprozess verloren gehen. Es ist nicht schwer, dieses Problem zu lösen, man muss es sich nur bewusst machen.

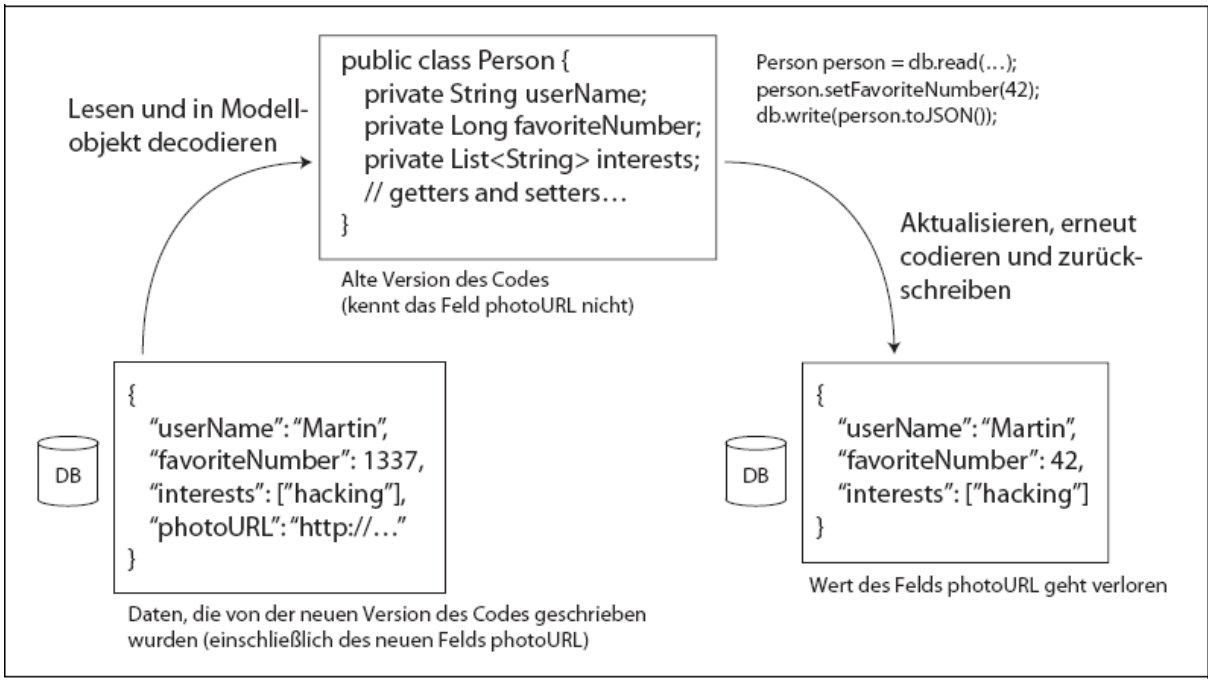



*Abbildung 4-7: Aktualisiert eine ältere Version der Anwendung Daten, die von einer neueren Version der Anwendung geschrieben wurden, können Daten verloren gehen, wenn Sie nicht aufpassen*.

## Verschiedene Werte, die zu verschiedenen Zeitpunkten geschrieben wurden

Eine Datenbank erlaubt es im Allgemeinen, jeden Wert jederzeit zu aktualisieren. Das heißt, dass es innerhalb derselben Datenbank Werte gibt, die vor fünf Millisekunden geschrieben wurden, und andere Werte, die seit fünf Jahren dort stehen.

Wenn Sie eine neue Version Ihrer Anwendung (zumindest einer serverseitigen Anwendung) bereitstellen, können Sie die alte Version innerhalb weniger Minuten vollkommen durch die neue Version ersetzen. Für den Datenbankinhalt gilt das nicht: Die fünf Jahre alten Daten werden immer noch da sein, und zwar in der ursprünglichen Codierung, sofern Sie sie seitdem nicht neu geschrieben haben. Diese Beobachtung wird manchmal zusammengefasst zu *Daten überdauern Code*.

Es ist sicherlich möglich, Daten in ein neues Schema zu schreiben (zu *migrieren*), doch insbesondere bei großen Datenmengen ist das sehr aufwendig, sodass die meisten Datenbanken dies nach Möglichkeit vermeiden. Die meisten relationalen Datenbanken erlauben einfache Schemaänderungen wie zum Beispiel das Hinzufügen einer neuen Spalte mit dem Standardwert `null`, ohne vorhandene Daten neu schreiben zu müssen.[5] Wenn eine alte Zeile gelesen wird,

füllt die Datenbank alle Spalten, die den codierten Daten von der Festplatte fehlen, mit `null`-Werten. Die Dokumentdatenbank Espresso von LinkedIn verwendet Avro zum Speichern und kann somit die Regeln zur Schemaevolution von Avro verwenden [23].

Durch Schemaevolution kann also die gesamte Datenbank so erscheinen, als wäre sie mit einem einzigen Schema codiert worden, selbst wenn die zugrunde liegende Speicherung Datensätze enthält, die mit verschiedenen historischen Versionen des Schemas codiert wurden.

## Archivspeicher

Vielleicht erzeugen Sie von Zeit zu Zeit einen Snapshot Ihrer Datenbank, etwa für Sicherungszwecke oder für das Laden in ein Data-Warehouse (siehe Abschnitt »Data-Warehousing« auf Seite 98). In diesem Fall wird der Daten-Dump in der Regel mit dem neuesten Schema codiert, auch wenn die ursprüngliche Codierung in der Quelldatenbank eine Mischung von Schemaversionen aus verschiedenen Zeiträumen enthält. Da Sie die Daten ohnehin kopieren, können Sie genauso gut die Kopie der Daten einheitlich codieren.

Wenn der Daten-Dump in einem Zug geschrieben wird und danach unveränderlich ist, sind Dateiformate wie das Avro-Objektcontainerformat passend. Dies ist auch eine gute Gelegenheit, um die Daten in ein analysefreundliches spaltenorientiertes Format wie zum Beispiel Parquet (siehe Abschnitt »Spaltenkomprimierung« auf 105) zu codieren.

In Kapitel 10 gehen wir näher auf die Verwendung von Daten in Archivspeichern ein.

## Datenfluss über Dienste: REST und RPC

Wenn Prozesse über ein Netzwerk kommunizieren müssen, gibt es verschiedene Methoden, diese Kommunikation einzurichten. Am häufigsten geschieht dies über zwei Rollen: *Clients* und *Server*. Die Server machen über das Netzwerk eine API zugänglich, und die Clients können sich mit den Servern verbinden, um Anfragen an diese API zu stellen. Die vom Server zugänglich gemachte API wird als *Dienst* (*Service*) bezeichnet.

Das Web funktioniert auf diese Weise: Clients (Webbrowser) stellen Anfragen an Webserver, wobei sie GET-Anfragen ausführen, um HTML, CSS, JavaScript, Bilder usw. herunterzuladen, und POST-Anfragen, um Daten an den Server zu

übermitteln. Die API besteht aus einem standardisierten Satz von Protokollen und Datenformaten (HTTP, URLs, SSL/TLS, HTML usw.). Da sich Webbrowser, Webserver und Websiteautoren größtenteils an diese Standards halten, können Sie mit jedem Webbrowser auf jede Website zugreifen (zumindest in der Theorie!).

Webbrowser sind nicht der einzige Clienttyp. Zum Beispiel kann eine native App, die auf einem mobilen Gerät oder einem Desktopcomputer läuft, auch Netzwerkanfragen an einen Server stellen, und eine clientseitige JavaScript-Anwendung, die in einem Webbrowser läuft, kann per XMLHttpRequest zu einem HTTP-Client werden (eine als *Ajax* bekannte Technik [30]). In diesem Fall besteht die Antwort des Servers typischerweise nicht aus HTML, das für einen Menschen angezeigt wird, sondern aus Daten in einer Codierung, die für die weitere Verarbeitung durch den Code der clientseitigen Anwendung (wie zum Beispiel JSON) zweckmäßig ist. Obwohl HTTP als Transportprotokoll dienen kann, ist die API, die darauf aufsetzt, anwendungsspezifisch. Zudem müssen sich der Client und der Server über die Details dieser API einig sein.

Darüber hinaus kann ein Server selbst ein Client für einen anderen Dienst sein (zum Beispiel fungiert ein typischer Webappserver als Client für eine Datenbank). Oftmals gliedert man nach diesem Konzept eine große Anwendung in kleinere Dienste nach ihrem Funktionalitätsbereich, beispielsweise dass ein Dienst eine Anfrage an einen anderen stellt, wenn er eine bestimmte Funktionalität oder Daten von diesem anderen Dienst benötigt. Dieses Erstellen von Anwendungen hat man ursprünglich als *dienstorientierte Architektur* (Service-Oriented Architecture, SOA) bezeichnet, mittlerweile verfeinert und in das Architekturmuster *Microservices* umbenannt [31, 32].

In gewisser Hinsicht sind Dienste Datenbanken ähnlich: Typischerweise ermöglichen sie Clients, Daten zu übermitteln und abzufragen. Während aber Datenbanken beliebige Abfragen mit den Abfragesprachen ermöglichen, wie sie Kapitel 2 vorgestellt hat, machen Dienste eine anwendungsspezifische API zugänglich, die nur Eingaben und Ausgaben erlaubt, die von der Geschäftslogik (dem Anwendungscode) des Diensts vorbestimmt sind [33]. Diese Einschränkung bietet einen gewissen Grad der Kapselung: Dienste können mit fein abgestimmten Einschränkungen spezifizieren, was Clients tun dürfen und was nicht.

Ein wichtiges Designziel einer dienstorientierten oder auf Microservices aufgebauten Architektur ist es, die Anwendung leichter ändern und warten zu können, indem sich Dienste unabhängig voneinander bereitstellen und

erweitern lassen. Zum Beispiel sollte jeder Dienst einem Team gehören, und dieses Team sollte in der Lage sein, neue Versionen des Diensts herauszubringen, ohne sie mit anderen Teams abstimmen zu müssen. Mit anderen Worten sollten wir davon ausgehen, dass alte und neue Versionen von Servern und Clients gleichzeitig laufen, und deshalb müssen die von Servern und Clients codierten Daten über verschiedene Versionen der Dienst-API kompatibel sein – genau das, worüber wir in diesem Kapitel gesprochen haben.

## Webdienste

Wenn HTTP als zugrunde liegendes Protokoll für die Kommunikation mit dem Dienst verwendet wird, spricht man von einem *Webservice* (Webdienst). Vielleicht ist diese Bezeichnung nicht ganz zutreffend, weil Webservices nicht nur im Web, sondern auch in anderen Kontexten anzutreffen sind. Zum Beispiel:

1. Eine Clientanwendung, die auf dem Gerät eines Benutzers läuft (zum Beispiel eine native App auf einem mobilen Gerät oder eine JavaScript-Webapp, die Ajax verwendet) und Anfragen an einen Dienst über HTTP richtet. Diese Anfragen laufen typischerweise über das öffentliche Internet.
2. Ein Dienst, der Anfragen an einen anderen Dienst richtet, der zur selben Organisation gehört, sich oftmals innerhalb desselben Rechenzentrums befindet, als Teil einer dienstorientierten oder auf Microservices aufbauenden Architektur. (Software, die derartige Einsatzfälle unterstützt, wird auch *Middleware* genannt.)
3. Ein Dienst, der Anfragen an einen Dienst richtet, der zu einer anderen Organisation gehört, normalerweise über das Internet. Diese Methode dient dem Datenaustausch zwischen den Backendsystemen verschiedener Organisationen. In diese Kategorie fallen öffentliche APIs, die von Onlinediensten bereitgestellt werden, wie zum Beispiel Kreditkartenverarbeitungssysteme oder OAuth für gemeinsamen Zugriff auf Benutzerdaten.

Für Webservices gibt es zwei populäre Ansätze: REST und SOAP. Von der Philosophie her gesehen sind sie fast diametral entgegengesetzt und oftmals Gegenstand hitziger Debatten unter ihren jeweiligen Befürwortern.[6]

REST ist kein Protokoll, sondern eine Entwurfsphilosophie, die auf den Prinzipien von HTTP aufbaut [34, 35]. Im Vordergrund stehen einfache Datenformate, die Ressourcen über URLs identifizieren und HTTP-Features für

Cache-Steuerung, Authentifizierung und Aushandlung des Inhaltstyps verwenden. Verglichen mit SOAP, erfreut sich REST zunehmender Beliebtheit, zumindest im Rahmen organisationsübergreifender Dienstintegration [36], und hat oftmals mit Microservices zu tun [31]. Eine API, die nach den Prinzipien von REST konzipiert ist, wird als *RESTful* bezeichnet.

Im Unterschied dazu ist SOAP ein XML-basiertes Protokoll, um Netzwerk-APIAnfragen auszuführen.[7] Obwohl es meistens über HTTP verwendet wird, soll es unabhängig von HTTP sein und möglichst ohne HTTP-Features auskommen. Stattdessen bringt es eine ausgedehnte und komplexe Vielfalt verwandter Standards mit (das als *WS-** bekannte *Webservices-Framework*), die verschiedene Features hinzufügen [37].

Die API eines SOAP-Webservice wird mit einer XML-basierten Sprache namens WSDL (Web Services Description Language) beschrieben. WSDL ermöglicht die Codeerzeugung, sodass ein Client auf einen Remotedienst über lokale Klassen und Methodenaufrufe zugreifen kann (die in XML-Nachrichten codiert und durch das Framework wieder decodiert werden). In statisch typisierten Programmiersprachen ist dies zwar nützlich, aber nicht in dynamisch typisierten (siehe Abschnitt »Codeerzeugung und dynamisch typisierte Sprachen« auf Seite 135).

Da WSDL nicht als Klartextsprache ausgelegt ist und SOAP-Nachrichten oftmals zu komplex sind, um sie von Hand zu konstruieren, verlassen sich die Benutzer von SOAP stark auf Tool-Unterstützung, Codeerzeugung und IDEs [38]. Für Benutzer von Programmiersprachen, die von SOAP-Anbietern nicht unterstützt werden, ist die Integration mit SOAP-Diensten schwierig.

Auch wenn SOAP und die verschiedenen Erweiterungen angeblich standardisiert sind, verursacht die Interoperabilität zwischen den Implementierungen der verschiedenen Anbieter oftmals Probleme [39]. Aus allen diesen Gründen wird SOAP zwar immer noch in vielen größeren Unternehmen verwendet, ist aber in den meisten kleineren Firmen in Ungnade gefallen.

RESTful APIs bevorzugen eher einfachere Ansätze, bei denen in der Regel auch weniger Codegenerierung und automatisierte Tools erforderlich sind. Mit einem Definitionsformat wie zum Beispiel OpenAPI, das auch als Swagger [40] bekannt ist, lassen sich RESTful APIs beschreiben und Dokumentation produzieren.

## Die Probleme mit Remoteprozeduraufrufen (RPCs)

Webservices sind lediglich die neueste Inkarnation einer langen Reihe von Techniken, API-Anfragen über ein Netzwerk auszuführen. Viele dieser Techniken haben einen Hype erfahren, weisen aber ernsthafte Probleme auf. Enterprise JavaBeans (EJB) und RMI (Remote Method Invocation) von Java sind auf Java beschränkt. Das Distributed Component Object Model (DCOM) setzt Microsoft-Plattformen voraus. Die Common Object Request Broker Architecture (CORBA) ist äußerst komplex und bietet weder Abwärts- noch Vorwärtskompatibilität [41].

Alle diese Techniken beruhen auf dem *Remoteprozeduraufruf*-Konzept (Remote Procedure Call, RPC), das es schon seit den 1970er-Jahren gibt [42]. Das RPC-Modell versucht, eine Anfrage an einen entfernten Netzwerkdienst so aussehen zu lassen, als würden Sie in Ihrer Programmiersprache eine Funktion oder Methode innerhalb desselben Prozesses aufrufen (diese Abstraktion wird als *Ortstransparenz* bezeichnet). Obwohl RPC auf den ersten Blick komfortabel aussieht, ist der Ansatz grundlegend mangelhaft [43, 44]. Eine Netzwerkanfrage unterscheidet sich deutlich von einem lokalen Funktionsaufruf:

- Ein lokaler Funktionsaufruf ist vorhersagbar, und entweder verläuft er erfolgreich, oder er scheitert, was nur von Parametern abhängt, die unter Ihrer Kontrolle stehen. Eine Netzwerkanfrage ist nicht vorhersagbar: Die Anfrage oder Antwort kann aufgrund eines Netzwerkproblems verloren gehen, oder vielleicht ist der Remotecomputer langsam oder nicht verfügbar. Auf derartige Probleme haben Sie überhaupt keinen Einfluss. Da Netzwerkprobleme häufig vorkommen, müssen Sie darauf vorbereitet sein, zum Beispiel indem Sie eine gescheiterte Anfrage wiederholen.
- Ein lokaler Funktionsaufruf gibt entweder ein Ergebnis zurück, löst eine Ausnahme aus oder kehrt niemals zurück (weil er in eine Endlosschleife eintritt oder der Prozess abstürzt). Bei einer Netzwerkanfrage gibt es noch eine Möglichkeit: Sie kann infolge eines *Timeouts* ohne Ergebnis zurückkehren. In diesem Fall wissen Sie einfach nicht, was passiert ist: Wenn Sie keine Antwort vom Remotedienst erhalten, haben Sie keine Möglichkeit, herauszufinden, ob die Anfrage durchgekommen ist oder nicht. (Auf diesen Punkt gehen wir ausführlich in Kapitel 8 ein.)
- Wenn Sie eine gescheiterte Netzwerkanfrage erneut auslösen, ist es durchaus möglich, dass die vorherige Anfrage tatsächlich durchgekommen und nur die Antwort verloren gegangen ist. In diesem Fall verursacht die

Wiederholung, dass die Aktion mehrere Male ausgeführt wird, sofern Sie keinen Mechanismus für Deduplizierung (*Idempotenz*) in das Protokoll eingebaut haben. Bei lokalen Funktionsaufrufen stellt sich dieses Problem gar nicht. (Kapitel 11 beschäftigt sich ausführlicher mit Idempotenz.)

- Jeder Aufruf einer lokalen Funktion benötigt ungefähr die gleiche Zeit für die Ausführung. Eine Netzwerkanfrage ist wesentlich langsamer als ein Funktionsaufruf, und ihre Latenz streut ebenso breit: Unter günstigen Umständen ist sie in weniger als einer Millisekunde erledigt, doch wenn das Netzwerk von Datenstau betroffen oder der Remotedienst überlastet ist, kann es viele Sekunden dauern, um genau die gleiche Aufgabe abzuschließen.
- Wenn Sie eine lokale Funktion aufrufen, können Sie ihr effizient Referenzen (Zeiger) auf Objekte im lokalen Arbeitsspeicher übergeben. Bei einer Netzwerkanfrage müssen alle diese Parameter in eine Bytefolge codiert werden, die sich über das Netzwerk übertragen lässt. Das ist leicht möglich, wenn es sich bei den Parametern um einfache Datentypen wie Zahlen oder Strings handelt, wird aber schnell problematisch bei größeren Objekten.
- Da der Client und der Dienst in verschiedenen Programmiersprachen implementiert sein können, muss das RPC-Framework Datentypen von einer Sprache in eine andere übersetzen. Das kann hässlich enden, denn nicht alle Sprachen haben die gleichen Typen – denken Sie nur an die Probleme von JavaScript mit Zahlen größer als $2^{53}$ (siehe Abschnitt »JSON, XML und binäre Varianten« auf Seite 122). In einem einzelnen Prozess, der in einer einzigen Sprache geschrieben ist, gibt es dieses Problem nicht.

Alle diese Faktoren bedeuten, dass es keinen Grund gibt, einen Remotedienst zu sehr wie ein lokales Objekt in Ihrer Programmiersprache aussehen zu lassen, weil es sich um etwas grundsätzlich anderes handelt. Die Attraktivität von REST liegt zum Teil darin, dass dieses Programmierparadigma nicht versucht, die Tatsache zu verbergen, dass es ein Protokoll ist (obwohl das Leute scheinbar nicht davon abhält, RPC-Bibliotheken auf REST aufzubauen).

## Quo vadis RPC?

Trotz all dieser Probleme verschwindet RPC nicht. Auf allen in diesem Kapitel beschriebenen Codierungen hat man verschiedene RPC-Frameworks aufgebaut: Zum Beispiel bringen Thrift und Avro integrierte RPC-Unterstützung mit, gRPC ist

eine RPC-Implementierung, die Protocol Buffers verwendet, Finagle setzt ebenfalls auf Thrift, und Rest.li verwendet JSON über HTTP.

Diese neue Generation von RPC-Frameworks berücksichtigt klarer die Tatsache, dass sich eine Remoteanfrage von einem lokalen Funktionsaufruf unterscheidet. Zum Beispiel kapseln Finagle und Rest.li mithilfe von *Futures* (*Promises*) asynchrone Aktionen, die scheitern können. Zudem vereinfachen Futures Situationen, in denen Sie Anfragen an mehrere Dienste parallel stellen und deren Ergebnisse kombinieren müssen [45]. Das Protokoll gRPC unterstützt Streams, bei denen ein Aufruf nicht nur aus einer Anfrage und einer Antwort besteht, sondern aus einer Reihe von Anfragen und Antworten über eine gewisse Zeitspanne [46].

Einige dieser Frameworks bieten auch eine Diensterkennung (Service Discovery) – ermöglichen also einem Client herauszufinden, bei welcher IP-Adresse und Portnummer ein bestimmter Dienst zu finden ist. Auf dieses Thema kommen wir in Abschnitt »Anfragen weiterleiten« auf Seite 227 zurück.

Spezialisierte RPC-Protokolle mit einem binären Codierungsformat erreichen mitunter eine bessere Performance als generische Methoden wie JSON über REST. Allerdings bietet eine RESTful API andere wichtige Vorteile: Sie eignet sich gut zum Experimentieren und Debuggen (man kann ohne jegliche Codegenerierung oder Softwareinstallation Anfragen an sie über einen Webbrowser oder das Befehlszeilentool `curl` auslösen), sie wird von allen gängigen Programmiersprachen und Plattformen unterstützt, und es steht ein riesiges Ökosystem von Tools zur Verfügung (Server, Caches, Lastenausgleicher, Proxys, Firewalls, Überwachung, Debugging Tools, Testtools usw.)

Aus diesen Gründen scheint REST der vorherrschende Stil für öffentliche APIs zu sein. Der Schwerpunkt von RPC-Frameworks liegt auf Anfragen zwischen Diensten, die zur selben Organisation gehören und sich typischerweise innerhalb desselben Rechenzentrums befinden.

## Datencodierung und Evolution für RPC

Für die Evolvierbarkeit ist es wichtig, dass RPC-Clients und -Server unabhängig voneinander geändert und bereitgestellt werden können. Verglichen mit dem Datenfluss durch Datenbanken (wie im letzten Abschnitt beschrieben) können wir eine vereinfachende Annahme im Fall von Datenfluss durch Dienste treffen: Es ist vernünftig, anzunehmen, dass zuerst alle Server und danach alle Clients

aktualisiert werden. Somit brauchen Sie nur Abwärtskompatibilität für Anfragen und Vorwärtskompatibilität für Antworten.

Die Eigenschaften von Abwärts- und Vorwärtskompatibilität eines RPC-Schemas werden von der jeweils verwendeten Codierung geerbt:

- Thrift, gRPC (Protocol Buffers) und Avro RPC lassen sich nach den Kompatibilitätsregeln des jeweiligen Codierungsformats evolvieren.
- In SOAP werden Anfragen und Antworten mit XML-Schemas spezifiziert. Diese lassen sich evolvieren, doch es gibt einige heikle Stolpersteine [47].
- RESTful APIs verwenden meistens JSON (ohne ein formal spezifiziertes Schema) für Antworten und JSON- oder URI-codierte/Formular-codierte Parameter für Anfragen. Das Hinzufügen optionaler Anfrageparameter und neuer Felder zu Antwortobjekten betrachtet man normalerweise als Änderungen, die Kompatibilität bewahren.

Dienstkompatibilität ist ein schwieriges Thema aufgrund der Tatsache, dass RPC oftmals für die Kommunikation über organisatorische Grenzen hinweg verwendet wird, sodass der Provider eines Diensts oftmals keine Kontrolle über seine Clients hat und sie nicht zu einem Upgrade zwingen kann. Somit muss Kompatibilität für einen langen Zeitraum aufrechterhalten werden, möglicherweise auf unbestimmte Zeit. Ist eine Änderung erforderlich, die die Kompatibilität verletzt, wird der Provider letzten Endes oftmals mehrere Versionen der Dienst-API nebeneinander verwalten.

Es gibt keine Übereinkunft darüber, wie die API-Versionsverwaltung funktionieren soll (d.h., wie ein Client anzeigen kann, welche Version der API er verwenden möchte [48]). Bei RESTful APIs verwendet man üblicherweise eine Versionsnummer in der URL oder im HTTP-`Accept`-Header. Bei Diensten, die einen bestimmten Client mit API-Schlüsseln identifizieren, kann man die von einem Client angeforderte API-Version auf dem Server speichern und für diese ausgewählte Version die Aktualisierung über eine separate Verwaltungsschnittstelle zulassen [49].

## Datenfluss beim Nachrichtenaustausch

Wir haben uns die verschiedenen Methoden angesehen, wie codierte Daten von einem Prozess zu einem anderen fließen. Bisher ging es um REST und RPC (wobei ein Prozess eine Anfrage über das Netzwerk an einen anderen Prozess sendet und so schnell wie möglich eine Antwort erwartet) sowie Datenbanken

(bei denen ein Prozess codierte Daten schreibt und ein anderer Prozess sie zu einem späteren Zeitpunkt liest).

In diesem letzten Abschnitt werfen wir einen kurzen Blick auf Systeme zum *asynchronen Nachrichtenaustausch*, die zwischen RPC und Datenbanken angesiedelt sind. Sie sind RPC in dem Sinne ähnlich, dass die Anfrage eines Clients (normalerweise als *Nachricht* bezeichnet) an einen anderen Prozess mit geringer Latenz zugestellt wird. Die Ähnlichkeit mit Datenbanken kommt daher, dass die Nachricht nicht über eine direkte Netzwerkverbindung gesendet wird, sondern über einen Vermittler geht, einen sogenannten *Nachrichtenbroker* (auch als *Nachrichtenwarteschlange* oder *nachrichtenorientierte Middleware* bezeichnet), der die Nachricht vorübergehend speichert.

Die Verwendung eines Nachrichtenbrokers hat mehrere Vorteile verglichen mit direktem RPC:

- Er kann als Puffer agieren, wenn der Empfänger nicht verfügbar oder überlastet ist. Somit verbessert er die Systemzuverlässigkeit.
- Er kann Nachrichten automatisch erneut an einen Prozess senden, der abgestürzt ist. Er verhindert damit, dass Nachrichten verloren gehen.
- Er vermeidet es, dass der Sender die IP-Adresse und Portnummer des Empfängers kennen muss (was vor allem in einer Cloudumgebung nützlich ist, wo virtuelle Computer oftmals hinzugefügt und entfernt werden).
- Er erlaubt es, eine Nachricht an mehrere Empfänger gleichzeitig zu senden.
- Er entkoppelt den Sender logisch vom Empfänger (der Sender veröffentlicht Nachrichten einfach und kümmert sich nicht darum, wer sie konsumiert).

Allerdings besteht der Unterschied zu RPC darin, dass die Kommunikation per Nachrichtenaustausch in der Regel nur in einer Richtung verläuft: Ein Sender erwartet normalerweise nicht, eine Antwort auf seine Nachrichten zu empfangen. Zwar kann ein Prozess eine Antwort senden, doch geschieht das in der Regel in einem eigenen Kanal. Dieses Kommunikationsmuster ist *asynchron*: Der Sender wartet nicht darauf, dass die Nachricht zugestellt wird, sondern sendet sie einfach und vergisst sie dann.

## Nachrichtenbroker

In der Vergangenheit wurde die Landschaft der Nachrichtenbroker dominiert von kommerzieller Unternehmenssoftware von Firmen wie TIBCO, IBM

WebSphere und webMethods. In jüngerer Zeit sind Open-Source-Implementierungen wie RabbitMQ, ActiveMQ, HornetQ, NATS und Apache Kafka populär geworden. In Kapitel 11 vergleichen wir sie ausführlicher.

Die genaue Zustellungssemantik variiert je nach Implementierung und Konfiguration, doch im Allgemeinen werden Nachrichtenbroker wie folgt verwendet: Der eine Prozess sendet eine Nachricht an eine benannte *Warteschlange* (engl. queue) oder an ein Anmelde-Versendesystem (*Topic*), und der Broker stellt sicher, dass die Nachricht an einen oder mehrere *Verbraucher* bzw. *Abonnenten* oder *Konsumenten* dieser Warteschlange oder des Topics zugestellt wird. Es kann viele Produzenten und viele Verbraucher desselben Topics geben.

Ein Topic bietet Datenfluss nur in einer Richtung. Ein Verbraucher kann aber selbst Nachrichten an ein anderes Topic veröffentlichen (sodass Sie sie miteinander verketten können, wie Kapitel 11 zeigt) oder an eine Antwort-Warteschlange, die vom Sender der ursprünglichen Nachricht konsumiert wird (was einen Anfrage-Antwort-Datenfluss ähnlich zu RPC ermöglicht).

Nachrichtenbroker setzten in der Regel kein bestimmtes Datenmodell voraus – eine Nachricht ist lediglich eine Bytefolge mit bestimmten Metadaten, sodass Sie jedes Codierungsformat verwenden können. Wenn die Codierung abwärts- und vorwärtskompatibel ist, haben Sie die größte Flexibilität, um Erzeuger und Verbraucher unabhängig voneinander zu ändern und sie in beliebiger Reihenfolge zu aktualisieren.

Wenn ein Verbraucher Nachrichten an ein anderes Topic erneut veröffentlicht, müssen Sie darauf achten, unbekannte Felder zu bewahren, um das weiter oben im Kontext von Datenbanken beschriebene Problem (Abbildung 4-7) zu verhindern.

## Verteilte Aktorenframeworks

Das *Aktorenmodell* ist ein Programmiermodell für Nebenläufigkeit in einem einzelnen Prozess. Anstatt sich direkt mit Threads zu befassen (und den damit verbundenen Problemen wie Race Conditions, Sperren und Deadlocks), wird die Logik in *Aktoren* gekapselt. Jeder Aktor stellt typischerweise einen Client oder eine Entität dar, kann über einen bestimmten lokalen Zustand verfügen (der nicht mit einem anderen Aktor geteilt wird) und kommuniziert mit anderen Aktoren, indem asynchrone Nachrichten gesendet und empfangen werden. Die Nachrichtenzustellung ist nicht garantiert: In bestimmten Fehlerszenarien gehen

Nachrichten verloren. Da jeder Aktor zeitgleich nur eine Nachricht verarbeitet, braucht er sich nicht um Threads zu kümmern, und das Framework kann jedem Aktor unabhängig von den anderen Rechenzeit zuweisen.

In *verteilten Aktorenframeworks* dient dieses Programmiermodell dazu, eine Anwendung über mehrere Rechner zu skalieren. Der gleiche Mechanismus zum Nachrichtenaustausch wird verwendet, unabhängig davon, ob sich Sender und Empfänger auf demselben oder auf verschiedenen Knoten befinden. Wenn sie auf verschiedenen Knoten untergebracht sind, wird die Nachricht transparent in eine Bytefolge codiert, über das Netzwerk gesendet und auf der anderen Seite decodiert.

Ortstransparenz funktioniert im Aktorenmodell besser als in RPC, weil das Aktorenmodell von vornherein annimmt, dass Nachrichten verloren gehen können, selbst innerhalb eines einzelnen Prozesses. Obwohl die Latenz über das Netzwerk wahrscheinlich höher als innerhalb desselben Prozesses ist, gibt es weniger grundlegende Diskrepanzen zwischen lokaler und remoter Kommunikation, wenn das Aktorenmodell verwendet wird.

Im Prinzip integriert ein verteiltes Aktorenframework einen Nachrichtenbroker und das Aktorenprogrammiermodell in einem einzelnen Framework. Wenn Sie jedoch Rolling Upgrades Ihrer Aktor-basierten Anwendung durchführen wollen, müssen Sie sich dennoch um Vorwärts- und Abwärtskompatibilität kümmern, da Nachrichten von einem Knoten, der die neue Version ausführt, an einen Knoten, der die alte Version ausführt, gesendet werden können und umgekehrt.

Drei populäre verteilte Aktorenframeworks gehen wie folgt mit Nachrichtencodierungen um:

- *Akka* greift standardmäßig auf die integrierte Serialisierung von Java zurück, die weder Vorwärts- noch Abwärtskompatibilität bietet. Allerdings können Sie sie durch etwas wie Protocol Buffers ersetzen und damit gleichzeitig die Möglichkeit für Rolling Upgrades schaffen [50].
- *Orleans* unterstützt Rolling-Upgrade-Bereitstellungen mit einem eigenen Versionierungsmechanismus. Dieser erlaubt es, neue Methoden für Aktoren zu definieren (d.h. neue Nachrichtentypen, die ein Aktor verarbeiten kann) und dabei Abwärtskompatibilität zu bewahren, sofern existierende Methoden nicht geändert werden [51, 52].
- In *Erlang OTP* ist es überraschend schwer, Änderungen an Datensatzschemas vorzunehmen (obwohl das System viele Features besitzt, die für hohe Verfügbarkeit ausgelegt sind); Rolling Upgrades sind möglich,

müssen aber sorgfältig geplant werden [53]. Ein experimenteller neuer Datentyp `maps` (eine JSON-artige Struktur, die 2014 in Erlang R17 eingeführt wurde) könnte dies in Zukunft erleichtern [54].

# Zusammenfassung

In diesem Kapitel haben wir mehrere Methoden beschrieben, um Datenstrukturen in Bytefolgen für die Netzwerkübertragung oder das Speichern auf Festplatte umzuwandeln. Wir haben auch gezeigt, wie die Details dieser Codierungen nicht nur ihre Effizienz beeinflussen, sondern vor allem auch die Architektur der Anwendungen und Ihre Möglichkeiten für deren Evolution.

Insbesondere müssen viele Dienste Rolling Upgrades unterstützen, bei denen eine neue Version eines Diensts nach und nach auf jeweils wenige Knoten bereitgestellt wird, anstatt sämtliche Knoten gleichzeitig zu aktualisieren. Durch Rolling Upgrades lassen sich neue Versionen eines Diensts veröffentlichen, ohne dass Stillstandszeiten auftreten (was häufige kleine Änderungen gegenüber seltenen großen begünstigt). Zudem werden Bereitstellungen weniger riskant (es wird dadurch ermöglicht, fehlerhafte Versionen zu erkennen und ihre Veröffentlichung rückgängig zu machen, bevor sie eine große Anzahl von Benutzern betreffen). Diese Eigenschaften sind von großem Vorteil für die *Evolvierbarkeit* – Änderungen an einer Anwendung lassen sich leicht durchführen.

Bei Rolling Upgrades oder aus verschiedenen anderen Gründen müssen wir davon ausgehen, dass verschiedene Knoten unterschiedliche Versionen unseres Anwendungscodes ausführen. Somit ist es wichtig, sämtliche Daten, die im System unterwegs sind, in einer Weise zu codieren, die Abwärtskompatibilität (neuer Code kann alte Daten lesen) und Vorwärtskompatibilität (alter Code kann neue Daten lesen) bietet.

Wir haben mehrere Formate der Datencodierung und ihre Kompatibilitätseigenschaften untersucht:

- Programmiersprachenspezifische Codierungen sind auf eine einzige Programmiersprache eingeschränkt und bieten oftmals weder Vorwärts- noch Abwärtskompatibilität.
- Textformate wie JSON, XML und CSV sind weitverbreitet, und ihre Kompatibilität hängt davon ab, wie Sie sie verwenden. Sie bringen optionale Schemasprachen mit, die manchmal hilfreich und manchmal

hinderlich sind. Diese Formate sind bei Datentypen etwas unklar, sodass Sie etwa bei Zahlen und binären Strings genau aufpassen müssen.

- Binäre Formate, die von Schemas gesteuert werden, wie zum Beispiel Thrift, Protocol Buffers und Avro, ermöglichen eine kompakte, effiziente Codierung mit klar definierter Semantik für Vorwärts- und Abwärtskompatibilität. Die Schemas sind auch als Dokumentation und für Codegenerierung in statisch typisierten Sprachen nützlich. Allerdings haben diese Formate den Nachteil, dass sie erst decodiert werden müssen, bevor sie für den Menschen verständlich sind.

Wir haben auch mehrere Modi des Datenflusses erörtert und dabei verschiedene Szenarien veranschaulicht, in denen Datencodierungen wichtig sind:

- Datenbanken, wobei der Prozess, der in die Datenbank schreibt, die Daten codiert, und der Prozess, der aus der Datenbank liest, sie decodiert.
- RPC und REST APIs, wobei der Client eine Anfrage codiert, der Server die Anforderung decodiert, eine Antwort codiert, und der Client schließlich die Antwort decodiert.
- Asynchroner Nachrichtenaustausch (über Nachrichtenbroker oder Aktoren), wobei Knoten miteinander kommunizieren, indem sie sich Nachrichten schicken, die der Sender codiert und der Empfänger decodiert.

Schlussfolgernd lässt sich sagen, dass mit etwas Sorgfalt Abwärts- und Vorwärtskompatibilität sowie Rolling Upgrades durchaus realisierbar sind. Möge die Weiterentwicklung Ihrer Anwendung schnell vorankommen.

## Literaturverzeichnis


[1] *Java Object Serialization Specification*. docs.oracle.com, 2010.

[2] *Ruby 2.2.0 API Documentation*. ruby-doc.org, Dezember 2014.

[3] *The Python 3.4.3 Standard Library Reference Manual*. docs.python.org, Februar 2015.

[4] *EsotericSoftware/kryo*. github.com, Oktober 2014.

[5] *CWE-502: Deserialization of Untrusted Data*. Common Weakness Enumeration, cwe.mitre.org, 30. Juli 2014.

[6] Steve Breen: *What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability*. foxglovesecurity.com, 6. November 2015.

[7] Patrick McKenzie: *What the Rails Security Issue Means for Your Startup*. kalzumeus.com, 31. Januar 2013.

[8] Eishay Smith: *jvm-serializers wiki*. github.com, November 2014.

[9] *XML Is a Poor Copy of S-Expressions*. c2.com wiki.

[10] Matt Harris: *Snowflake: An Update and Some Very Important Information*. E-Mail an Twitter Development Talk mailing list, 19. Oktober 2010.

[11] Shudi (Sandy) Gao, C. M. Sperberg-McQueen und Henry S. Thompson: *XML Schema 1.1*. W3C Recommendation, Mai 2001.

[12] Francis Galiegue, Kris Zyp und Gary Court: *JSON Schema*. IETF Internet-Draft, Februar 2013.

[13] Yakov Shafranovich: *RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files*. October 2005.

[14] *MessagePack Specification*. msgpack.org.

[15] Mark Slee, Aditya Agarwal und Marc Kwiatkowski: Thrift: *Scalable Cross-Language Services Implementation*. Facebook technical report, April 2007.

[16] *Protocol Buffers Developer Guide*. Google, Inc., developers.google.com.

[17] Igor Anishchenko: *Thrift vs Protocol Buffers vs Avro – Biased Comparison*. slideshare.net, 17. September 2012.

[18] *A Matrix of the Features Each Individual Language Library Supports*. wiki.apache.org.

[19] Martin Kleppmann: *Schema Evolution in Avro, Protocol Buffers and Thrift*. martin.kleppmann.com, 5. Dezember 2012.

[20] *Apache Avro 1.7.7 Documentation*. avro.apache.org, Juli 2014.

[21] Doug Cutting, Chad Walters, Jim Kellerman, et al.: *[PROPOSAL] New Subproject: Avro*. E-Mail thread on hadoop-general mailing list, mail-archives.apache.org, April 2009.

[22] Tony Hoare: *Null References: The Billion Dollar Mistake*, auf der QCon London, März 2009.

[23] Aditya Auradkar und Tom Quiggle: *Introducing Espresso – LinkedIn's Hot New Distributed Document Store*. engineering.linkedin.com, 21. Januar 2015.

[24] Jay Kreps: *Putting Apache Kafka to Use: A Practical Guide to Building a Stream Data Platform (Part 2)*. blog.confluent.io, 25. Februar 2015.

[25] Gwen Shapira: *The Problem of Managing Schemas*. radar.oreilly.com, 4. November 2014.

[26] *Apache Pig 0.14.0 Documentation*. pig.apache.org, November 2014.

[27] John Larmouth: *ASN.1 Complete*. Morgan Kaufmann, 1999. – ISBN 978-0-122-33435-1

[28] Russell Housley, Warwick Ford, Tim Polk und David Solo: *RFC 2459: Internet X.509 Public Key Infrastructure: Certificate and CRL Profile*. IETF Network Working Group, Standards Track, Januar 1999.

[29] Lev Walkin: *Question: Extensibility and Dropping Fields*. lionet.info, 21. September 2010.

[30] Jesse James Garrett: *Ajax: A New Approach to Web Applications*. adaptivepath.com, 18. Februar 2005.

[31] Sam Newman: *Building Microservices*. O'Reilly Media, 2015. – ISBN 978-1-491-95035-7

[32] Chris Richardson: *Microservices: Decomposing Applications for Deployability and Scalability*. infoq.com, 25. Mai 2014.

[33] Pat Helland: *Data on the Outside Versus Data on the Inside*, auf der 2. Biennial Conference on Innovative Data Systems Research (CIDR), Januar 2005.

[34] Roy Thomas Fielding: *Architectural Styles and the Design of Network-Based Software Architectures*. PhD Thesis, University of California, Irvine, 2000.

[35] Roy Thomas Fielding: *REST APIs Must Be Hypertext-Driven*. roy.gbiv.com, 20. Oktober 2008.

[36] *REST in Peace, SOAP*. royal.pingdom.com, 15. Oktober 2010.

[37] *Web Services Standards as of Q1 2007*. innoq.com, Februar 2007.

[38] Pete Lacey: *The S Stands for Simple*. harmful.cat-v.org, 15. November 2006.

[39] Stefan Tilkov: *Interview: Pete Lacey Criticizes Web Services*. infoq.com, 12. Dezember 2006.

[40] *OpenAPI Specification (fka Swagger RESTful API Documentation Specification) Version 2.0*. swagger.io, 8. September 2014.

[41] Michi Henning: *The Rise and Fall of CORBA*. ACM Queue, Bd. 4, Nr. 5, S. 28–34, Juni 2006. doi:10.1145/1142031.1142044

[42] Andrew D. Birrell und Bruce Jay Nelson: *Implementing Remote Procedure Calls*. ACM Transactions on Computer Systems (TOCS), Bd. 2, Nr. 1, S. 39–59, Februar 1984. doi:10.1145/2080.357392

[43] Jim Waldo, Geoff Wyant, Ann Wollrath und Sam Kendall: *A Note on Distributed Computing*. Sun Microsystems Laboratories, Inc., Technical Report TR-94-29, November 1994.

[44] Steve Vinoski: *Convenience over Correctness*. IEEE Internet Computing, Bd. 12, Nr. 4, S. 89–92, Juli 2008. doi:10.1109/MIC.2008.75

[45] Marius Eriksen: *Your Server as a Function*. im 7. Workshop on Programming Languages and Operating Systems (PLOS), November 2013. doi: 10.1145/ 2525528.2525538

[46] *grpc-common Documentation*. Google, Inc., github.com, Februar 2015.

[47] Aditya Narayan und Irina Singh: *Designing and Versioning Compatible Web Services*. ibm.com, 28. März 2007.

[48] Troy Hunt: *Your API Versioning Is Wrong, Which Is Why I Decided to Do It 3 Different Wrong Ways*. troyhunt.com, 10. Februar 2014.

[49] *API Upgrades*. Stripe, Inc., April 2015.

[50] Jonas Bonér: *Upgrade in an Akka Cluster*. E-Mail an akka-user mailing list, grokbase.com, 28. August 2013.

[51] Philip A. Bernstein, Sergey Bykov, Alan Geller, et al.: *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Microsoft Research Technical Report MSR-TR-2014-41, März 2014.

[52] *Microsoft Project Orleans Documentation*. Microsoft Research, dotnet.github.io, 2015.

[53] David Mercer, Sean Hinde, Yinso Chen und Richard A O'Keefe: *beginner: Updating Data Structures*. E-Mail-Thread auf erlang-questions mailing list, erlang.com, 29. Oktober 2007.

[54] Fred Hebert: *Postscript: Maps*. learnyousomeerlang.com, 9. April 2014.

# Index

## Numerisch



## A

## B

## C

## D

## E

## F

## G

## H

## I

## J

## K

## L