

# CORBA als Ordnung in verteilten Anwendungen

«Essentials CORBA-Buch» von Andreas Sayegh  
Kapitel 3 (Ergänzung IIOP)

Anton Böhm

[anton.boehm@itServe.ch](mailto:anton.boehm@itServe.ch)

***itServe AG***

P.O.Box

Länggass-Str.26

CH-3000 Bern 9

Norbert Hranitzky

[norbert.hranitzky@mchp.siemens.de](mailto:norbert.hranitzky@mchp.siemens.de)

Tel. +41 31 305 16 16

# Inhalt

- CORBA-Protokolle
- GIOP-Protokoll
  - Common Data Representation (CDR)
  - Interoperable Object Reference (IOR)
  - Messages
  - Verbindungsverwaltung
- IIOP-Protokoll

# Übersicht

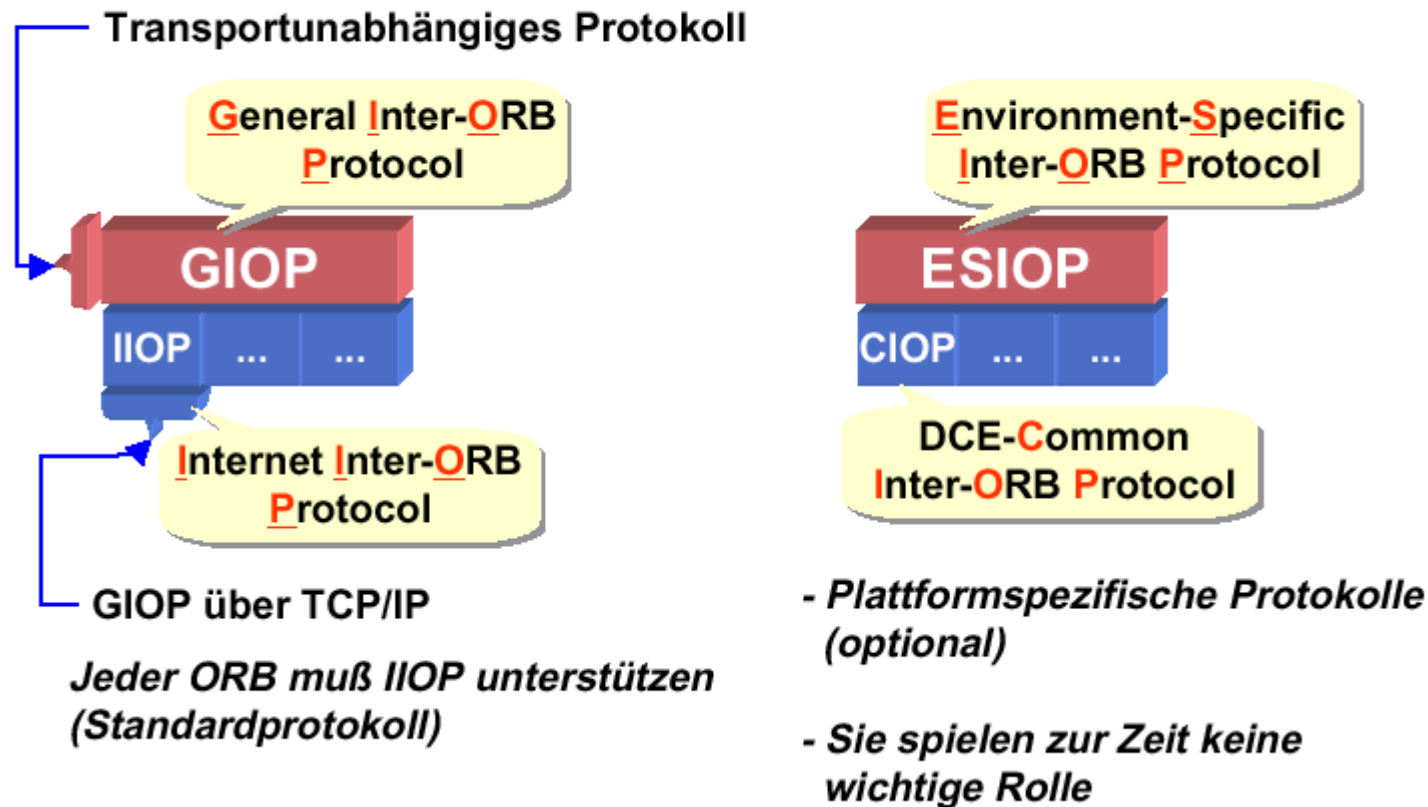
- IIOp ist das Standardprotokoll der CORBA-Architektur
- Jede CORBA-Implementierung muß IIOp unterstützen
- Beim Einsatz des CORBA Security Service wird das SECIOP-Protokoll verwendet
- In der Zukunft wird das Java-RMI (Remote Method Invocation) das IIOp-Protokoll auch unterstützen
- IIOp ist das Interoperabilitätsprotokoll für das Komponentenmodell Enterprise JavaBeans
- Die Firma Iona bietet eine portable Implementierung des IIOp-Protokolls an (IIOp Engine, in C implementiert)

# Kommunikationsprotokoll

*Was muß die Beschreibung eines  
Kommunikationsprotokolls enthalten?*

- Wie die Daten “über die Leitung” geschickt werden (sog. Marshaling oder Serialisierung)
- Welche Nachrichten ausgetauscht werden (“Ping-Pong”)
- Wie die Nachrichten aufgebaut werden
- Welche Eigenschaften das Transportprotokoll erfüllen muß
- Wie der Verbindungsaufbau bzw. Verbindungsabbau erfolgt

# CORBA-Protokolle



# GIOP-Protokoll

- Festlegung der externen Darstellung der IDL-Typen  
(CDR = Common Data Representation)
  - Berücksichtigung der Byte-Anordnung (big-endian oder little-endian): “Der Empfänger macht’s richtig”
- Festlegung der Nachrichtentypen und der Nachrichtenformate
- Voraussetzungen für den Transport:
  - Verbindungsorientiertes Protokoll
  - Partner müssen beim Verlust der Verbindung benachrichtigt werden
  - Der Transport ist zuverlässig (z.B. keine Änderung der Byte-Reihenfolge)
  - Asymmetrische Verbindungsnutzung
    - Request werden nur vom Client abgesetzt

# Common Data Representation

## *Einfache Datentypen*

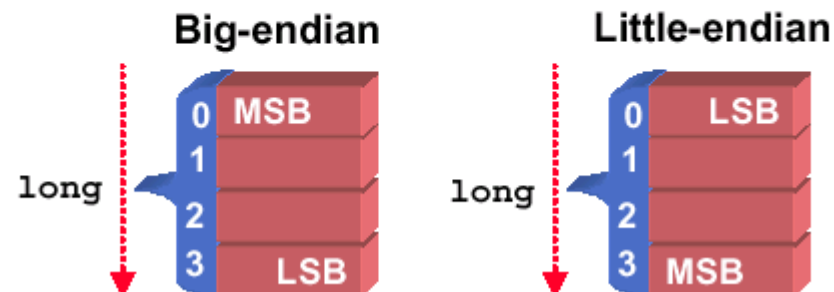
- Die einfachen Datentypen werden als eine Reihe von Octets dargestellt
- Die einfachen Datentypen werden folgermaßen ausgerichtet (die Ausrichtung entspricht der Anzahl der Octets)

char	1
wchar	1,2 oder 4 (abhängig vom CodeSet)
octet	1
(unsigned) short	2
(unsigned) long	4
(unsigned) long long	8
float	4
double	8
long double	8
boolean	1
enum	4

# Common Data Representation

## *Einfache Datentypen*

- Die Reihenfolge der Octets für die numerischen Datentypen kann **Big-endian** oder **Little-endian** sein (short, long, long long, float, double)
- Ein Indikator in der Message zeigt an, wie die numerischen Datentypen dargestellt werden. Der Empfänger muß ggf. die Daten konvertieren.
- Die Gleitkommazahlen werden im IEEE-Format dargestellt





# Common Data Representation

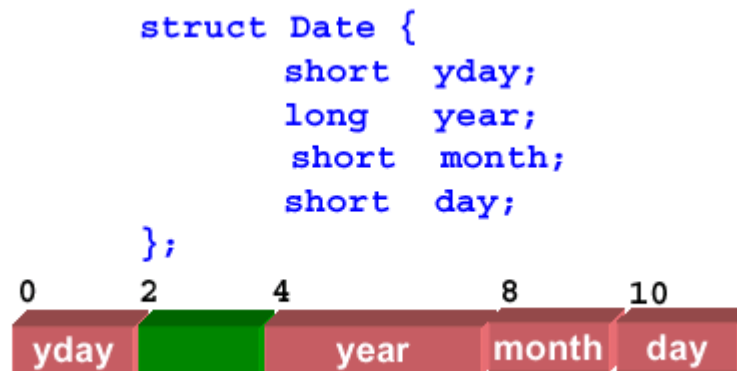
## *Einfache Datentypen*

- Ein **Octet** ist 8 Bit lang und wird unverändert übertragen
- Der **Boolean**-Typ hat der Wert 1 für TRUE und 0 für FALSE
- Die Darstellung der **Wide-Characters** (wchar) ist abhängig vom CodeSet, zum Beispiel:
  - bei ISO10646 UCS-2 als `unsigned short`
  - bei ISO10646 UCS-4 als `unsigned long`

# Common Data Representation

## *Zusammengesetzte Datentypen*

- Die Darstellung der Daten des Typs **struct** entspricht der Darstellung seiner Komponenten in der Reihenfolge der Deklaration :



# Common Data Representation

## *Zusammengesetzte Datentypen*

- **Arrays** werden als die Reihe ihrer Komponenten dargestellt (Arrays haben in IDL immer eine Länge). Bei mehrdimensionalen Arrays wächst der Index der ersten Dimension am langsamsten

```
short vector[4];
```



- Der Datentyp **sequence** wird kodiert als die Anzahl der Elemente (als unsigned long) gefolgt von den Elementen:

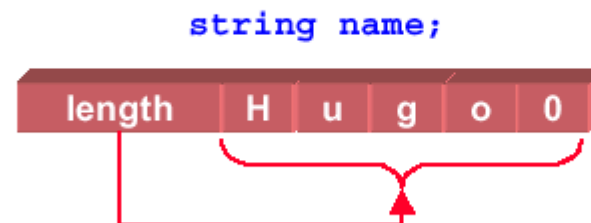
```
sequence<short> data;
```



# Common Data Representation

## *Zusammengesetzte Datentypen*

- Die Darstellung der **Strings** besteht aus der Länge des Strings als Octets (als unsigned long) gefolgt der Reihe der Zeichen als Octets. Die Länge und der Inhalt enthalten auch das Null-Zeichen

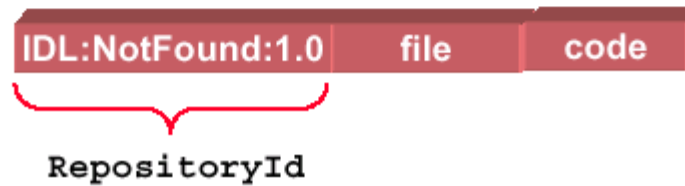


# Common Data Representation

## *Exceptions*

- Exceptions werden folgendermaßen kodiert:
  - RepositoryId als String
  - Der Inhalt der Exception als struct

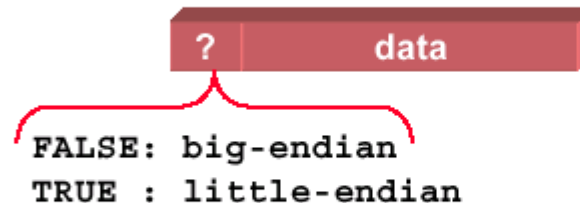
```
exception NotFound {  
    String file;  
    long   code;  
};
```



# Common Data Representation

## *Encapsulation*

- Einige GIOP-Protokollelemente werden als Octet-Strom eingeschalt. Das erste Byte (Boolean) zeigt die Byte-Anordnung der eingeschalteten Daten an:



- Üblicherweise werden die Daten als `sequence<octet>` kodiert

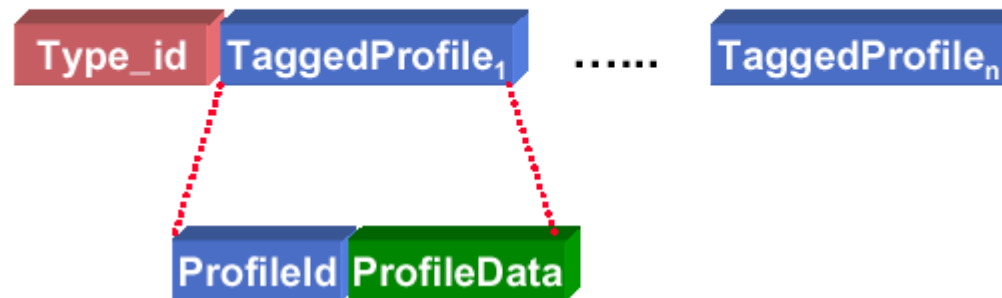
# Common Data Representation

## *Object values*

- In CORBA2.2 können Objekte nur per Referenz übergeben werden (d.h. sie müssen Remote-Objekte sein)
- In Zukunft können auch in CORBA Objekte als Wert übertragen werden (object-per-value), wie in Java. Zur Zeit ist der Standardisierungsprozeß noch nicht abgeschlossen
- In JDK1.2 wird dieser Mechanismus für die Unterstützung von IIOP für RMI benötigt

# IOR (Interoperable Object Reference)

- IOR ist die “Objektreferenz” eines CORBA-Objektes
- Die IOR besteht aus der Typeld und einer Liste von sog. TaggedProfiles
- Ein TaggedProfile enthält die Informationen, die für die Adressierung des Objektes für ein Protokoll erforderlich sind



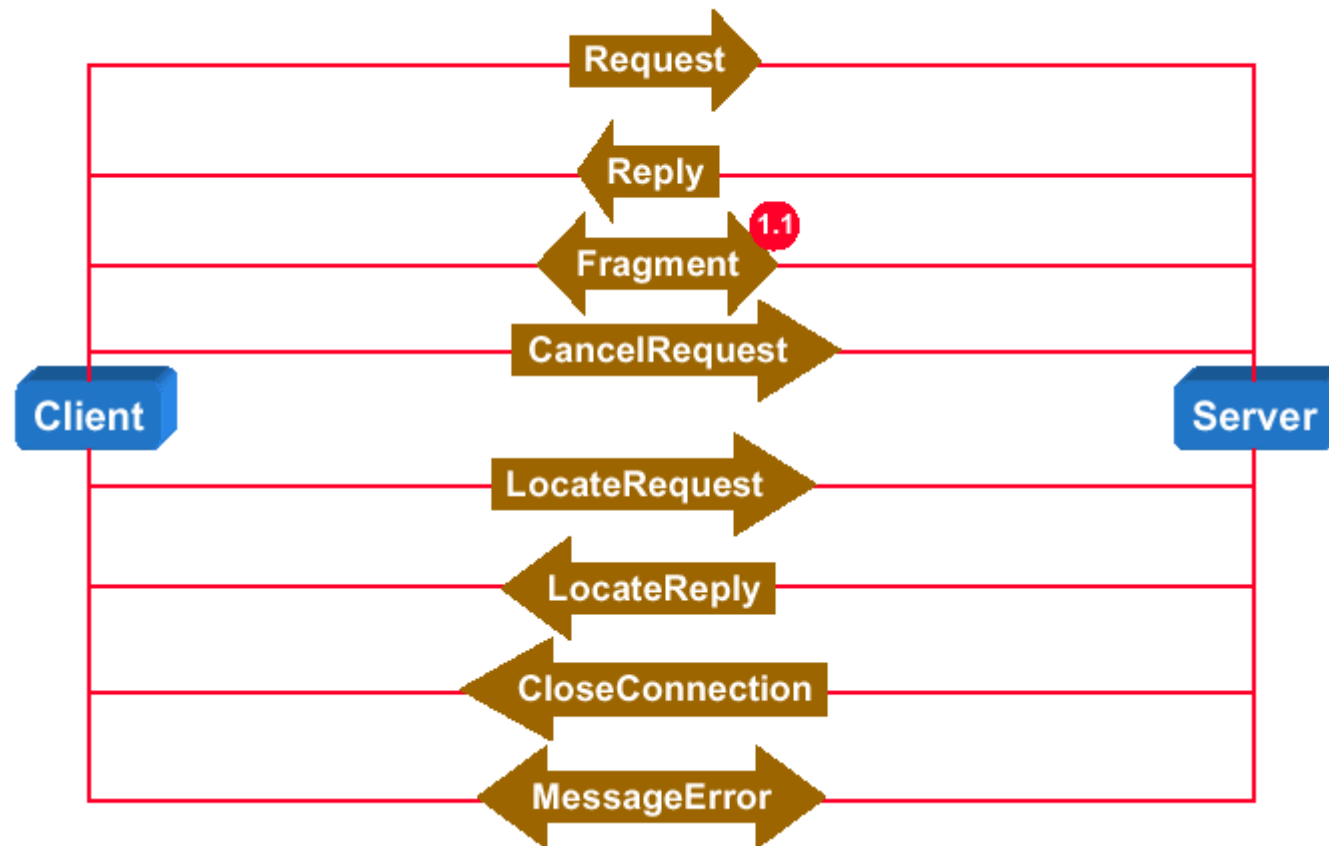


## IOR

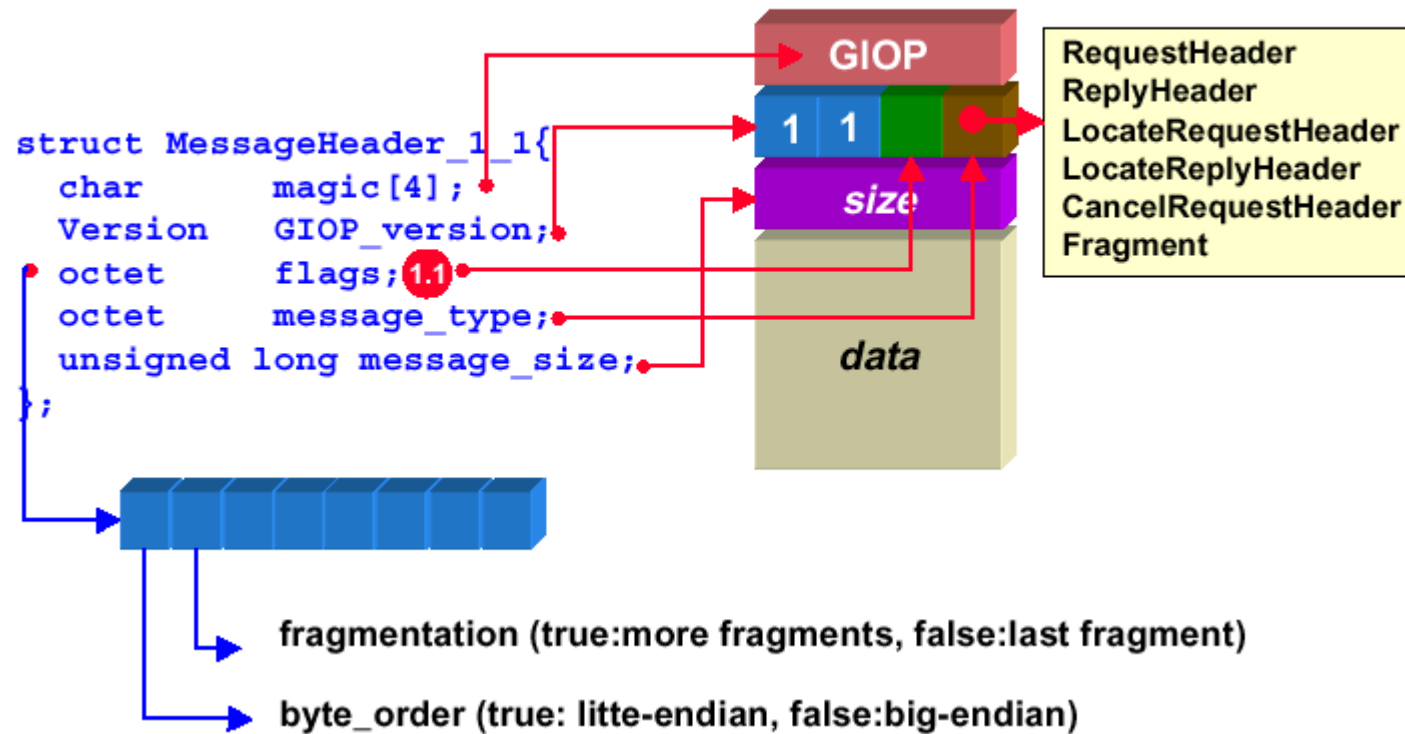
```
typedef unsigned long ProfileId;
const ProfileId TAG_INTERNET_IOP = 0;
const ProfileId TAG_MULTIPLE_COMPONENTS = 1;

struct TaggedProfile {
    ProfileId tag;
    sequence <octet> profile_data;
};
struct IOR {
    string type_id;
    sequence <TaggedProfile> profiles;
};
typedef unsigned long ComponentId;
struct TaggedComponent {
    ComponentId tag;
    sequence <octet> component_data;
};
typedef sequence <TaggedComponent> MultipleComponentProfile;
```

## GIOP Messages



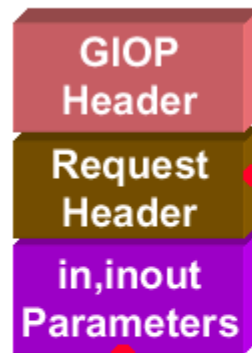
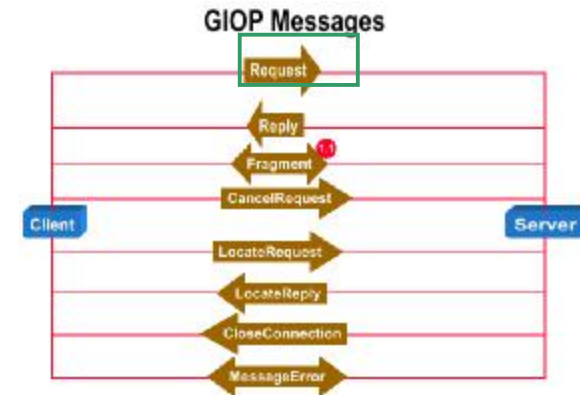
# GIOP-Message Header



## GIOP-Message Header

- Das Feld **magic** ('GIOP') dient zur Plausibilitätsprüfung
- Die aktuelle Protokollversion ist: 1.1 (**GIOP\_version**)  
(die Unterschiede zur Version 1.0 sind mit **1.1** gekennzeichnet)
- Das Feld **flags** enthält zwei Indikatoren:
  - die Byte-Anordnung zeigt an, wie die numerischen Daten kodiert sind (big-endian, little-endian)
  - die Fragmentierung zeigt an, ob weitere Folgenachrichten kommen, die zu diesem Aufruf gehören
- Das Feld **message\_type** legt den Typ der Nachricht fest
- Das Feld **message\_length** enthält die Länge der folgenden Daten

# Request



## ■ Protokollelement für die Methodenaufufe

```

struct RequestHeader_1_1{
    IOP:ServiceContextList  service_context;
    unsigned long    request_id;
    boolean          response_expected;
    1.1  octet        reserved[3];
    sequence<octet> object_key;
    string           operation;
    Principal        requesting_principal;
};
    
```

in- bzw. inout-parameter in der Reihenfolge der Deklaration

# RequestHeader

- Das Feld `service_context` enthält Service-spezifischer Daten (z.B. Transaktion, CodeSet):

```
typedef unsigned long ServiceId;

struct ServiceContext {

    ServiceId      context_id;
    sequence<octet> context_data;
};

typedef sequence<ServiceContext> ServiceContextList;

const ServiceId TransactionService = 0;
```

# RequestHeader

## Beispiel Transaktion: PropagationContext

```
Struct otid_t {
    long formatId: /* 0 is OSI-TP */
    long bqual_length;
    sequence<octet> tid;
};

struct TransIdentity {
    Coordinator coord;
    Terminator term;
    otid_t      otid;
}

struct PropagationContext {
    unsigned long      timeout;
    TransIdentity      current;
    sequence<TransIdentity> parents;
    any implementation_specific_data;
};
```

# RequestHeader

- Die `request_id` wird vom Client (genauer gesagt, vom Laufzeitsystem auf der Clientseite) erzeugt und dient zur Identifizierung des Requests (die Antwort muß die gleiche `request_id` enthalten)
- Wenn der Indikator `response_expected` gesetzt ist, erwartet der Client eine Antwort. Ist der Indikator nicht gesetzt, ist der Aufruf ein sog. oneway-Aufruf
- Das Feld `object_key` identifiziert das Objekt auf dem Server
- Das Feld `operation` enthält den Namen der Methode
- Das Feld `requesting_principal` enthält die Daten für die Identifikation des Aufrufers. Das Format für `Principal` ist undefiniert (es wird als `sequence<octet>` kodiert)



# RequestBody

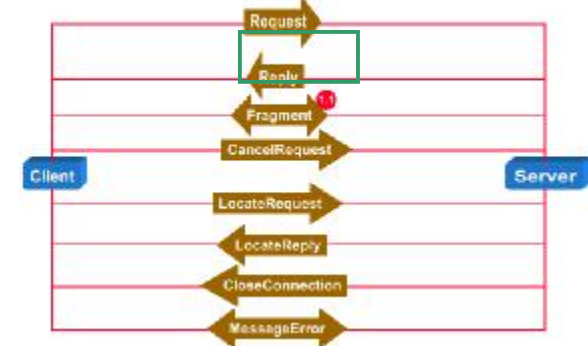
- Im **RequestBody** werden die Argumente (in und inout) kodiert

Beispiel:

```
void deposit(in string name, out long old, inout short state);
```



## GIOP Messages

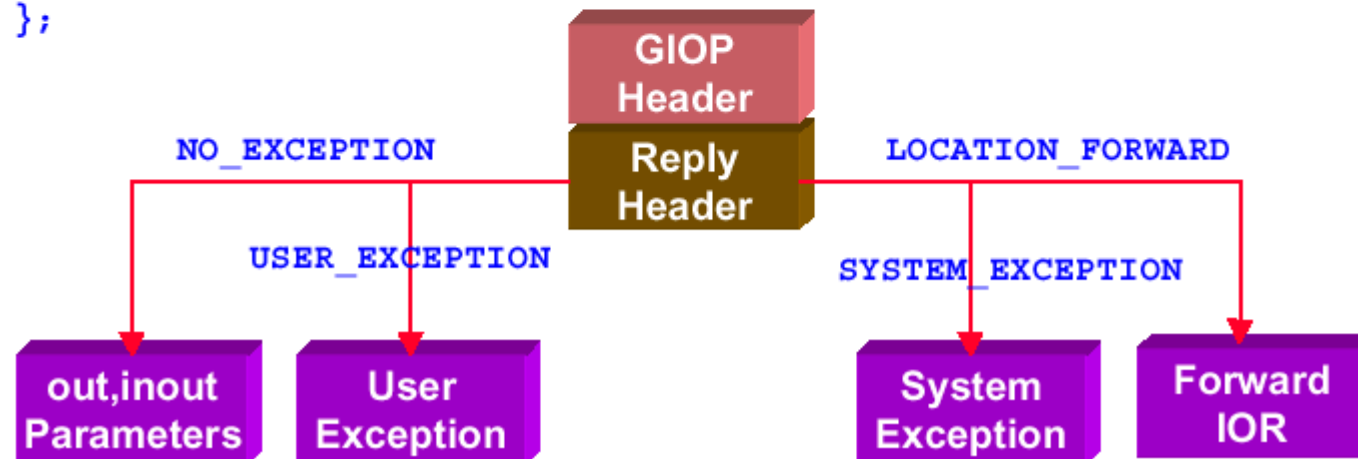


## Reply

Protokollelement für die Antwort auf einen Methodenaufruf

```

enum ReplyStatus {
    NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION, LOCATION_FORWARD
};
struct ReplyHeader {
    IOP:ServiceContextList service_context;
    unsigned long request_id;
    ReplyStatus reply_status;
};
    
```



# ReplyHeader

- Im Feld `service_context` liefert der Server die service-spezifischen Daten zurück
- Das Feld `request_id` enthält den gleichen Wert wie das entsprechende Feld im Request
- Das Feld `reply_status` zeigt an, ob
  - der Aufruf erfolgreich war (`NO_EXCEPTION`)
  - eine Exception aufgetreten ist (`..._EXCEPTION`)
  - der Aufruf an eine andere Objektreferenz geschickt werden soll (`LOCATION_FORWARD`)

# ReplyBody

`reply_status = NO_EXCEPTION`

- Der ReplyBody enthält den Rückgabewert und die Ausgabe-parameter (out, inout)

Beispiel:

```
long deposit(in string name, out long old);
```

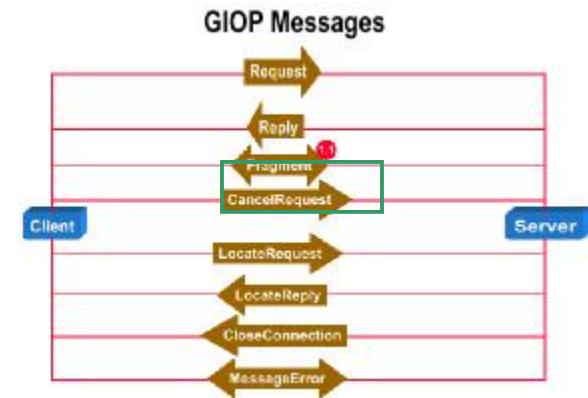


`reply_status = USER_EXCEPTION | SYSTEM_EXCEPTION`

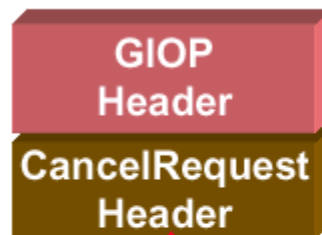
- Der ReplyBody enthält die Exception

`reply_status = LOCATION_FORWARD`

- Der ReplyBody enthält die neue Objektreferenz (IOR). Das Laufzeitsystem wird den Aufruf mit der neuen Objektreferenz wiederholen



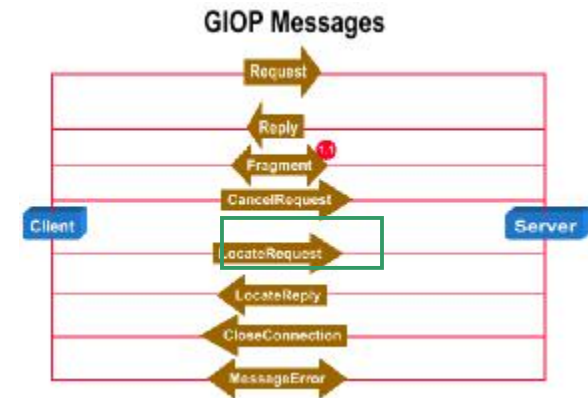
## CancelRequest



Protokollelement für das Abbrechen eines vorangegangenen Request- bzw. LocateRequest-Aufrufs

```

struct CancelRequestHeader {
    unsigned long request_id;
};
  
```



## LocateRequest

Protokollelement für die Lokalisierung eines Objektes



```

struct LocateRequestHeader {
    unsigned long    request_id;
    sequence<octet> object_key;
};
  
```

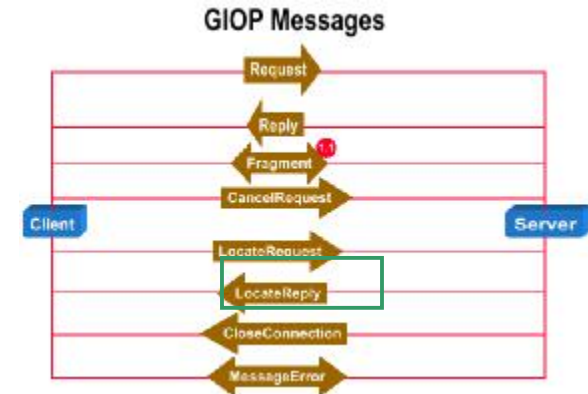
## LocateReply

Protokollelement für die Antwort auf den Lokalisierungsaufruf



```
enum LocateStatusType {
    UNKNOWN_OBJECT, OBJECT_HERE, OBJECT_FORWARD
};
struct LocateReplyHeader {
    unsigned long    request_id;
    LocateStatusType locate_status;
};
```

Forward  
IOR

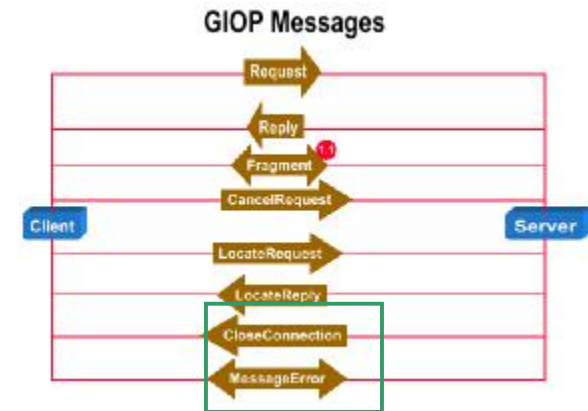


## LocateReply

■ Der Server antwortet auf den LocateRequest-Aufruf:

- **UNKNOWN\_OBJECT** : wenn er das Objekt nicht kennt
- **OBJECT\_HERE** : wenn er das Objekt kennt und dafür “zuständig” ist
- **OBJECT\_FORWARD** : wenn das Objekt eine andere Objektreferenz hat (z.B. auf einem anderen Rechner oder wenn der Server eine andere Portnummer hat)





## CloseConnection

GIOP  
Header

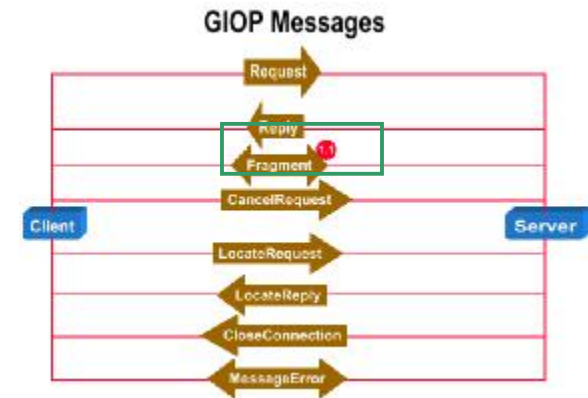
Der Server schließt die Verbindung

## MessageError

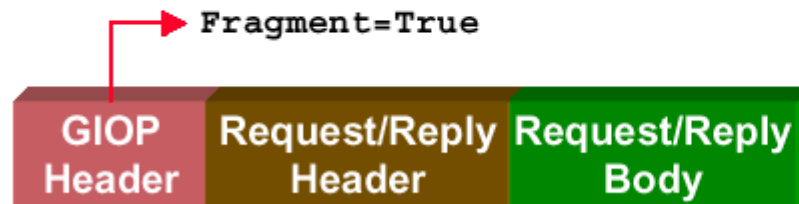
GIOP  
Header

Der Server oder der Client kann die Message nicht dekodieren (z.B. Magic oder Version falsch)

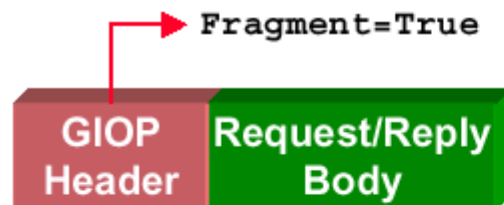
# Fragment 1.1



Folgenachricht bei Fragmentierung (für Request, Reply)

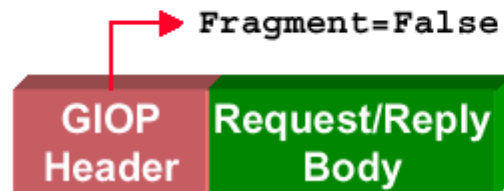


Erste Nachricht



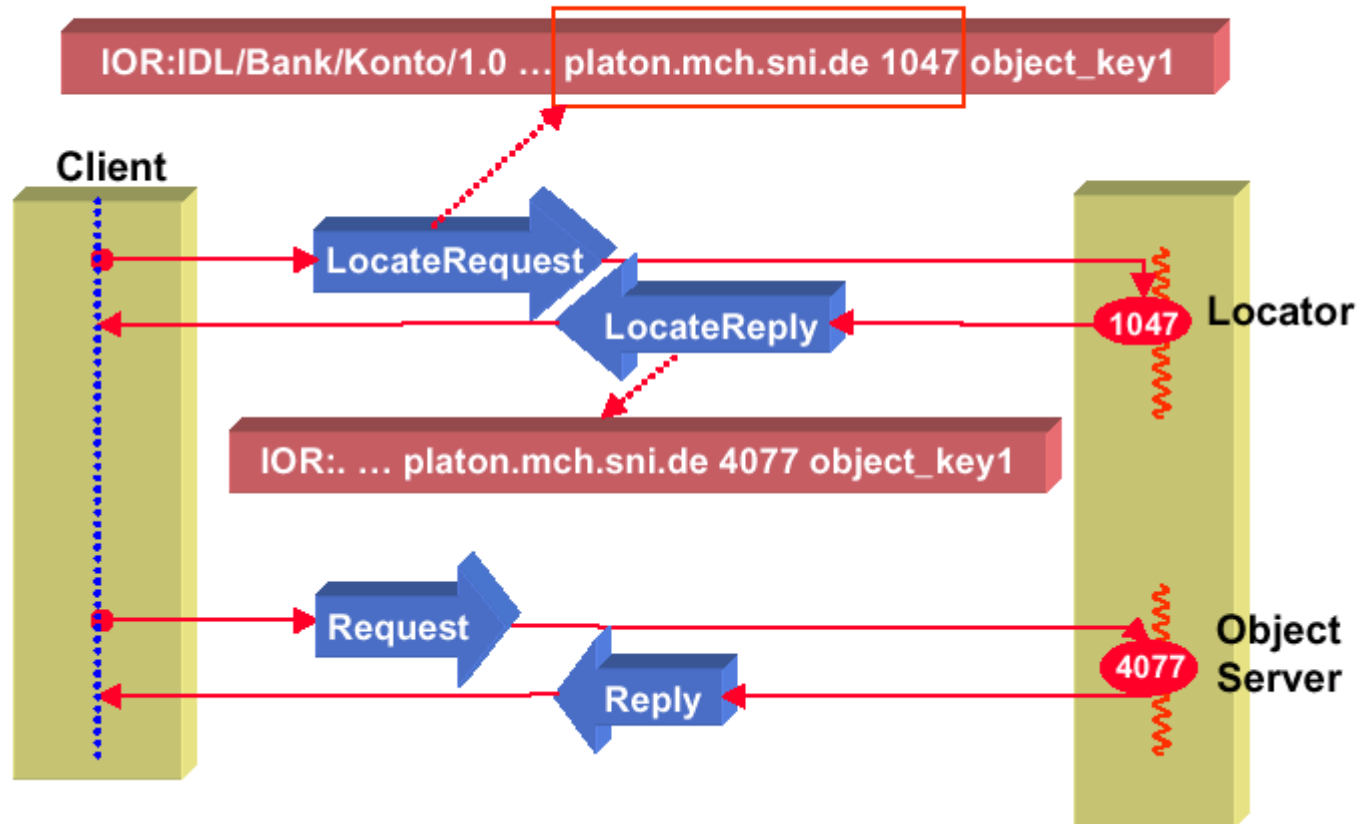
Nächste Nachricht(en)

.....



Letzte Nachricht

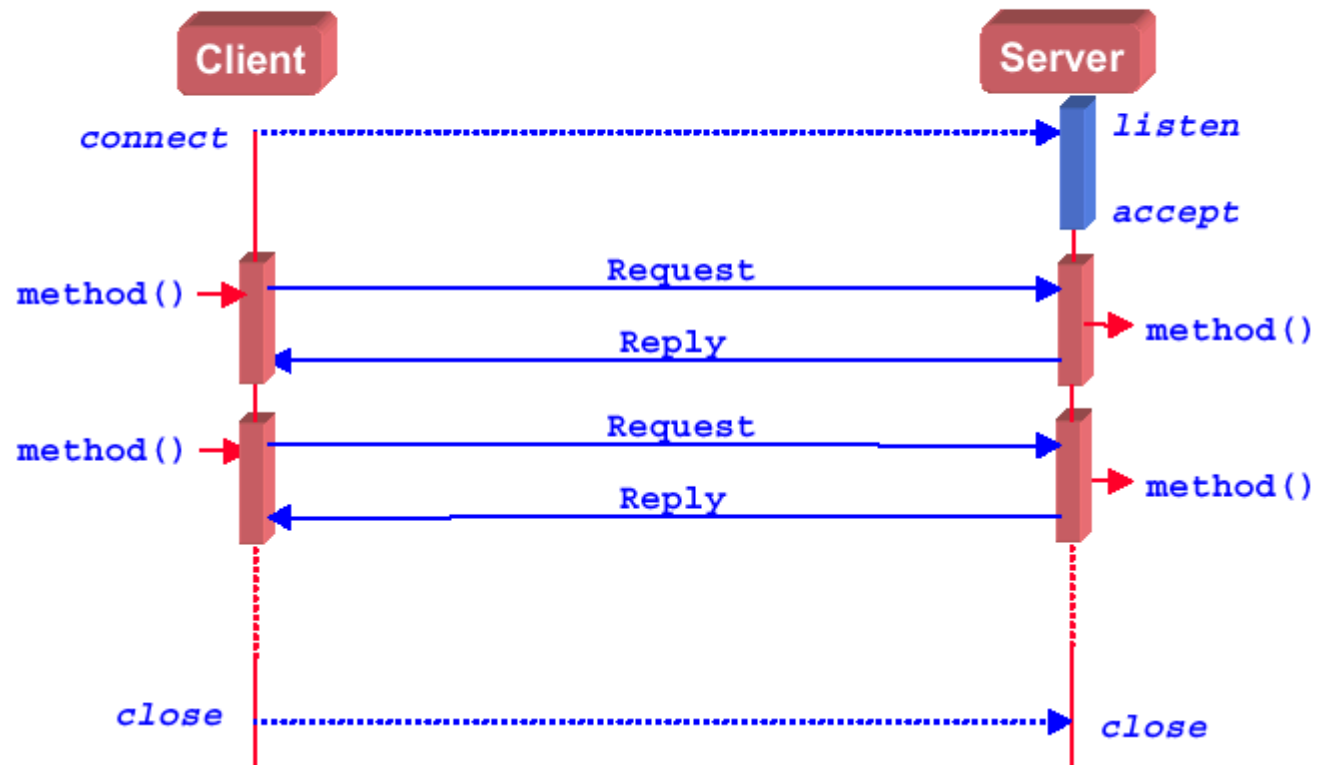
# Lokalisierung



# Verbindungsverwaltung

- Der Verbindungsaufbau wird immer vom Client initiiert. Der Client nutzt die Adressierungsinformationen in der Objektreferenz, um die Verbindung aufzubauen
- Requests werden nur vom Client abgesetzt
- Nur GIOP-Nachrichten werden über die GIOP-Verbindungen gesendet
- Der Client kann die Verbindung schließen
- Wenn die Verbindung vom Server ordnungsgemäß geschlossen werden soll, muß er die Message CloseConnection senden
- Über eine Verbindung können Aufrufe für mehrere Objekte abgesetzt werden

# Kommunikation



# IIOP

# IIOP-Protokoll

- IIOP ist das GIOP-Protokoll über TCP/IP
- Legt fest, wie die Objektreferenz aussieht (IIOP-IOR)



# IIOP-IOR

## Objektreferenz für das IIOP-Protokoll



```
const ProfileId TAG_INTERNET_IOP = 0;
```

```
struct ProfileBody {  
    Version  
    string  
    unsigned short  
    sequence<octet>  
};
```

```
    iiop_version:  
    host;  
    port;  
    object_key
```

Rechnername oder  
IP-Adresse

Portnummer

Identifiziert das Objekt  
auf dem Server-ORB

```
1.1 sequence<IOP::TaggedComponent> components;
```



## IIOP-IOR

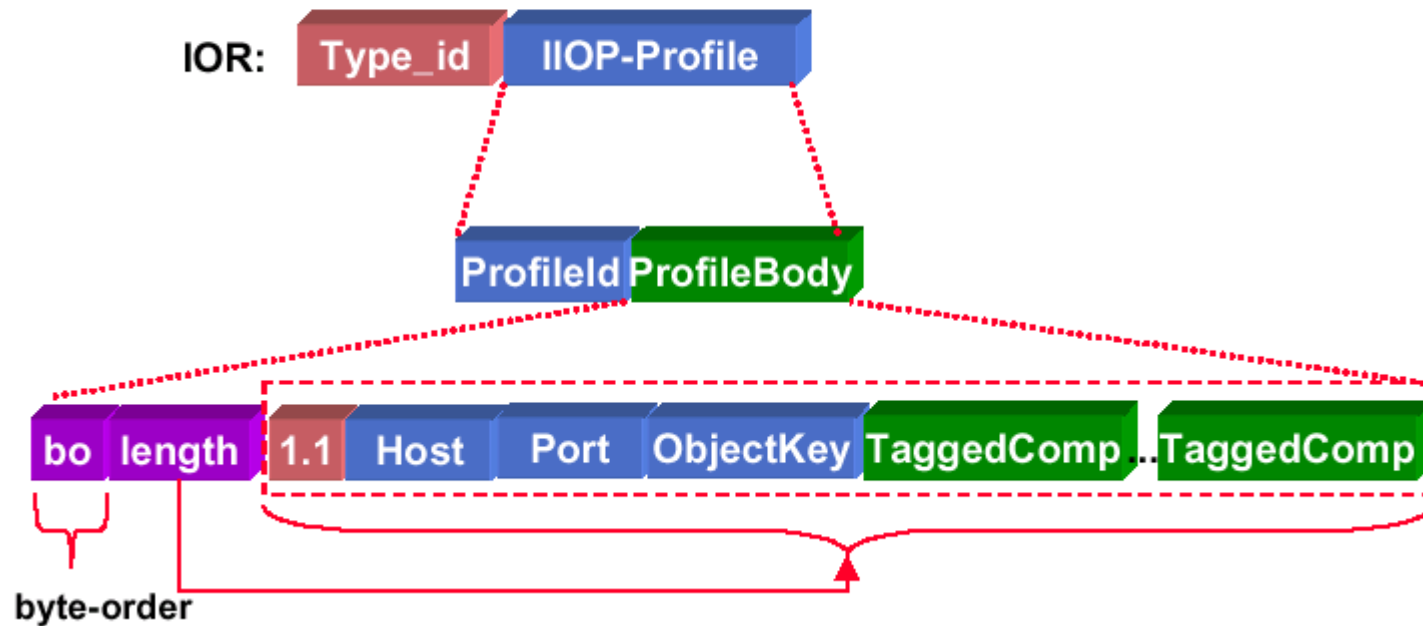
- Die TaggedComponents enthalten
  - Daten für Dienste (Security) oder
  - zusätzliche Informationen für den Aufruf
- Es gibt Standardkomponenten, die nie aus der IOR entfernt werden dürfen, wenn sie vorhanden sind:

```
const ComponentId TAG_ORB_TYPE = 0 ;  
const ComponentId TAG_CODE_SETS = 1 ;  
const ComponentId TAG_SEC_NAME = 14 ;  
const ComponentId TAG_ASSOCIATION_OPTIONS = 13 ;  
const ComponentId TAG_GENERIC_SEC_MECH = 12 ;
```

- Man kann eigene Komponenten definieren, aber sie können von einem ORB entfernt werden

# IIOP-IOR

- Der **ProfileBody** wird als “encapsulated octet stream” im Feld `profile_data` der IOR-Struktur kodiert



# Abkürzungen

CIOP	DCE <u>C</u> ommon <u>I</u> nter- <u>O</u> RB <u>P</u> rotocol
CORBA	<u>C</u> ommon <u>O</u> bject <u>R</u> equest <u>B</u> roker <u>A</u> rchitecture
ESIOP	<u>E</u> nvironment- <u>S</u> pecific <u>I</u> nter- <u>O</u> RB <u>P</u> rotocol
GIOP	<u>G</u> eneral <u>I</u> nter- <u>O</u> RB <u>P</u> rotocol
IIOP	<u>I</u> nternet <u>I</u> nter- <u>O</u> RB <u>P</u> rotocol
IDL	<u>I</u> nterface <u>D</u> efinition <u>L</u> anguage
IOR	<u>I</u> nteroperable <u>O</u> bject <u>R</u> eference
OMG	<u>O</u> bject <u>M</u> anagement <u>G</u> roup
ORB	<u>O</u> bject <u>R</u> equest <u>B</u> roker
RMI	<u>R</u> emote <u>M</u> ethod <u>I</u> nvocation
SECIOP	<u>S</u> ECure <u>I</u> nter- <u>O</u> RB <u>P</u> rotocol
SSL	<u>S</u> ecure <u>S</u> ocket <u>L</u> ayer
UCS	<u>U</u> niversal <u>C</u> haracter <u>S</u> et