

# Überblick

Fault-Tolerant CORBA Grundlagen

Mechanismen

Einstellung von Fehlertoleranzeigenschaften

Verwaltung von Replikaten

Fehlermanagement

Recoverymanagement

Implementierung

Zusammenfassung

# Literatur



## Object Management Group

CORBA/IIOP Specification (Chapter 23, Fault Tolerant CORBA)

*OMG Technical Committee Document formal/04-03-21, 2004.*



## Pascal Felber, Priya Narasimhan

Experiences, Strategies, and Challenges in Building  
Fault-Tolerant CORBA Systems

*IEEE Transactions on Computers, vol. 53, no. 5, pp. 497-511, 2004*



## R. Baldoni, C. Marchetti, R. Panella, L. Verde

Handling FT-CORBA Compliant Interoperable Object Group References

*WORDS '02: Proceedings of the The Seventh IEEE International Workshop  
on Object-Oriented Real-Time Dependable Systems, 2002*



## Priya Narasimhan

Fault-Tolerant CORBA: From Specification to Reality

*Computer , vol. 40, no. 1, pp.110-112, Jan. 2007*

# Wozu überhaupt Fehlertoleranz?

- ▶ Hardwarefehler
- ▶ Softwarefehler
- ▶ ständige Verfügbarkeit
  - ▶ Luftfahrt, Medizin, Anlagensteuerung, etc.
- ▶ Kosten bei Ausfall

## Ziel

*No single point of failure*

# Was ist ein Fehler?

## Fault (Fehlerursache)

- ▶ unerwünschter Zustand, der zu einem Fehler führen kann

## Error (Fehler)

- ▶ Systemzustand, der nicht den Spezifikationen entspricht

## Failure (Funktionsausfall)

- ▶ Dienstleistung ist nicht mehr möglich

## Singal/Shivaratri

An error is a manifestation of a fault in a system, which could lead to system failure.

# Klassifikation von Fehlern

## Gutmütige Fehler (benign faults)

- ▶ **Crash Stop:** Ein Knoten fällt komplett aus
  - ▶ **Fail Stop:** Jeder korrekte Knoten erfährt innerhalb endlicher Zeit vom Ausfall eines Knoten
  - ▶ **Fail Silent:** Keine perfekte Ausfallerkennung möglich (asynchrones oder partiell synchrones Modell)
- ▶ **Crash Recovery:** Ein korrekter Knoten kann endlich oft ausfallen und wiederanlaufen

## Bösartige Fehler (malicious faults)

- ▶ Häufig als **byzantinische** Fehler bezeichnet
- ▶ Fehlerhafte Prozesse können beliebige Aktionen ausführen, und dabei auch untereinander kooperieren

## FT-CORBA betrachtet **fail stop** Fehler

- ▶ Viele realen Systeme gehen von ausschließlich gutmütigen Fehlern aus. Dies ist aber nicht immer realistisch!

# Fehlertoleranz durch Replikation

## Redundanz

- ▶ erzeugen und verwalten von Kopien/Replikaten eines Objektes

## Erkennen von Fehlern

- ▶ erkennen das ein Prozeß oder Objekt ausgefallen ist

## Recovery

- ▶ ausgefallene Instanz ersetzen

# CORBA und FT-CORBA

## CORBA

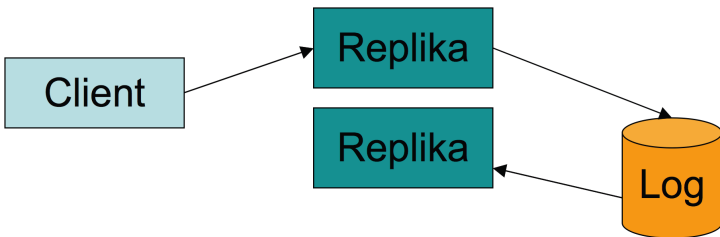
- ▶ CORBA spezifiziert keine Mechanismen für Redundanz
- ▶ Zuverlässige Verbindungen basierend auf TCP/IP ermöglichen beschränkte Erkennung von Ausfällen

## Ziele von FT-CORBA

- ▶ minimale Modifikation von Applikationen (sowohl auf Client- wie auch auf Serverseite)
- ▶ Unterstützung für Fehlertoleranz
  - ▶ verschiedene Fehlertoleranzanforderungen
  - ▶ verschiedene Applikationen
  - ▶ verschiedene Mechanismen zur Fehlererkennung und -behandlung

# Was muss eine Fehlertoleranzinfrastruktur leisten?

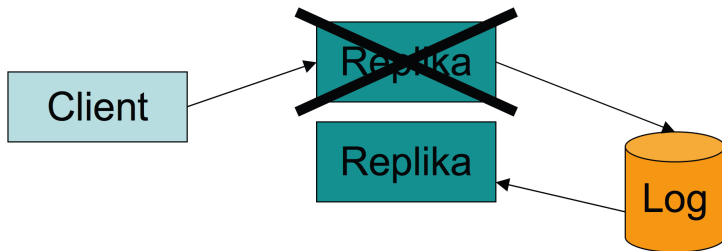
- ▶ Replikation von Objekten
- ▶ Erkennung von Ausfällen
- ▶ Logging von Anfragen und Zustandssicherung von Objektzuständen
- ▶ Zustandstransfer zur Initialisierung und Reinitialisierung von Objekten
- ▶ Transparente Verschattung von Ausfällen





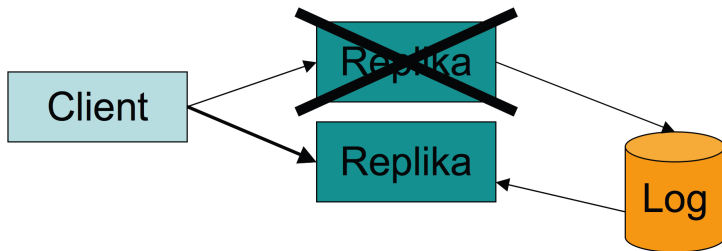
# Was muss eine Fehlertoleranzinfrastruktur leisten?

- ▶ Replikation von Objekten
- ▶ Erkennung von Ausfällen
- ▶ Logging von Anfragen und Zustandssicherung von Objektzuständen
- ▶ Zustandstransfer zur Initialisierung und Reinitialisierung von Objekten
- ▶ Transparente Verschattung von Ausfällen



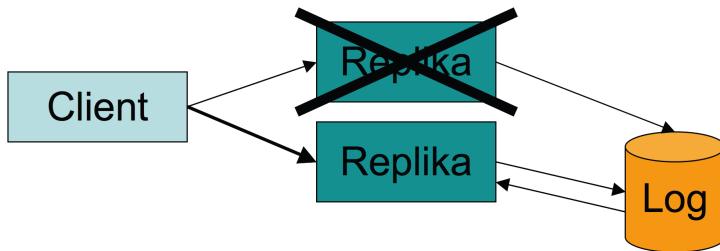
# Was muss eine Fehlertoleranzinfrastruktur leisten?

- ▶ Replikation von Objekten
- ▶ Erkennung von Ausfällen
- ▶ Logging von Anfragen und Zustandssicherung von Objektzuständen
- ▶ Zustandstransfer zur Initialisierung und Reinitialisierung von Objekten
- ▶ Transparente Verschattung von Ausfällen



# Was muss eine Fehlertoleranzinfrastruktur leisten?

- ▶ Replikation von Objekten
- ▶ Erkennung von Ausfällen
- ▶ Logging von Anfragen und Zustandssicherung von Objektzuständen
- ▶ Zustandstransfer zur Initialisierung und Reinitialisierung von Objekten
- ▶ Transparente Verschattung von Ausfällen



# Grundlagen

## Redundanz

- ▶ Objekte bilden die Einheit der Replikation
- ▶ Starke Konsistenz
  - ▶ Alle Replikate haben den gleichen Zustand
  - ▶ Ermöglicht einfache Anwendungsentwicklung
    - ▶ Client interagiert mit Objekten
  - ▶ Stellt hohe Anforderungen an die Mechanismen der Infrastruktur
    - ▶ Alle Mechanismen zur Fehlertoleranz werden durch die Middleware bereitgestellt

# Grundlagen

## Objektgruppe

- ▶ Menge aller Replikate eines Objektes bilden eine Objektgruppe
- ▶ Jede Gruppe verfügt über eine **Object Group Reference (IOR)**
- ▶ Zielsetzung
  - ▶ Replikationstransparenz
  - ▶ Fehlertransparenz

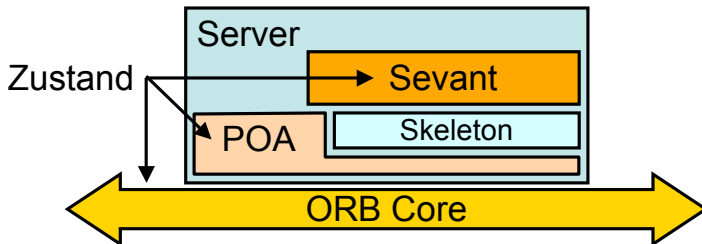
## Identität

- ▶ CORBA-Objekte werden durch ihren Ausführungsort identifiziert
  - ▶ schwache Form von Identität
- ▶ FT-CORBA benötigt eindeutige und ortsunabhängige Identifikation
  - ▶ Objektgruppen werden durch `FTDomainId` und `ObjectGroupId` identifiziert
  - ▶ einzelne Replikate durch `FTDomainId`, `ObjectGroupId` und Ort

# Grundlagen

## Zustand

- ▶ Zustand bildet die Menge an Informationen die zur Erhaltung von konsistenten Replikaten nötig ist
- ▶ Zustand ergibt sich durch das replizierte Objekt und die Infrastruktur (POA und ORB)



# Grundlagen

## Determinismus

- ▶ Replikation von Objekten erfordert deterministisches Verhalten
  - ▶ Aufrufe werden an alle Replikate in gleicher Reihenfolge zugestellt
  - ▶ Ausgehend von identischen Ausgangszuständen werden identische Endzustände erreicht
  - ▶ Objekt entspricht somit einem Zustandsautomat
- ▶ Problemfälle
  - ▶ objektexterne Informationen
    - ▶ z.B. Zeit, Zufallszahlen oder Aufruf an externen Informationsquellen
  - ▶ Koordinierung wenn parallel mehrere Threads ein Objekt verändern
    - ▶ z.B. Anforderung von Locks

# Grundlagen

## Replikationsart

- ▶ **Active** Replication
  - ▶ Replikate bearbeiten alle Anforderungen
- ▶ **Passive** Replication
  - ▶ Es gibt ein aktives Replikat (*master*) welches Aufrufe bearbeitet
  - ▶ Alle anderen Replikate sind in Wartestellung

## Anwendungen besitzen unterschiedliche Fehlertoleranzanforderungen

- ▶ Active Replication
  - ▶ Sehr kurze Verzögerungen im Fehlerfall
  - ▶ Hoher Aufwand unter anderem durch die erforderliche Gruppenkommunikation
- ▶ Passive Replication
  - ▶ Längere Verzögerung bei Ausfällen
  - ▶ Geringer Aufwand zur Laufzeit
  - ▶ Schnellere Antwortzeit im Normalbetrieb



# Grundlagen

## Zuständigkeiten Serverseite

- ▶ Objektreplikation
- ▶ Verwaltung der System- und Fehlertoleranzanforderungen pro Objektgruppe
  - ▶ Property Manager-Schnittstelle
- ▶ Erzeugen von replizierten Objekten
  - ▶ Generic Factory-Schnittstelle
  - ▶ Replication Manager-Schnittstelle
  - ▶ Zustandstransfer
- ▶ Erkennung von Ausfällen

# Grundlagen

## Zuständigkeiten Clientseite

- ▶ Failover
  - ▶ Falls ein Server nicht antwortet wird es entweder erneut versucht oder ein anderer Server wird kontaktiert
  - ▶ Aufrufe werden nur einmal durch das replizierte Objekt ausgeführt (muss durch die Serverseite unterstützt werden)
- ▶ Adressierung
  - ▶ Veraltete Referenzen werden in Kooperation mit dem Server ersetzt
    - ▶ Server liefert aktuelle Version
- ▶ Ausfall der Verbindung
  - ▶ z. B. kein Replikat ist erreichbar
  - ▶ Anwendung wird benachrichtigt

# Grundlagen

## Kontrolle und Verwaltung

- ▶ Infrastruktur kontrolliert Fehlertoleranz
  - ▶ Automatische Erzeugung von Replikaten
  - ▶ Automatische Erhaltung der Konsistenz
- ▶ Applikation kontrolliert Fehlertoleranz
  - ▶ Applikation lenkt die Erzeugung und Platzierung von Replikaten
  - ▶ Applikation gewährleistet die Konsistenz
  - ▶ Achtung: Nur in Spezialfällen nötig!

# Grundlagen

## Fault Tolerance Domain

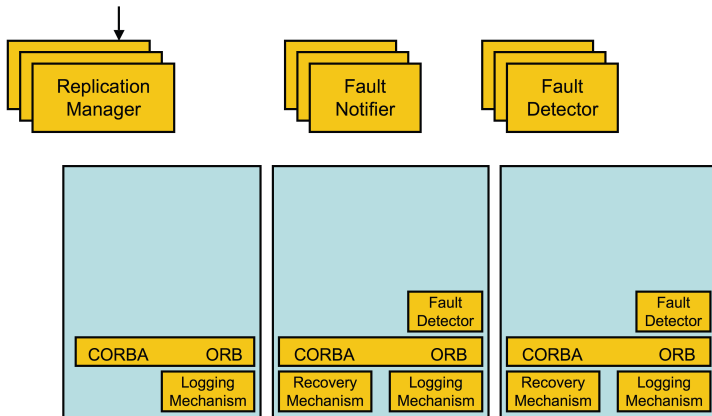
- ▶ Menge von Rechnern zur Bereitstellung von fehlertoleranten Applikationen
- ▶ Zentrale Komponente bildet der Replication Manager
  - ▶ steuert Erzeugung und Platzierung von Replikaten

# Limitationen des FT-CORBA Standards

- ▶ Determinismus bleibt Aufgabe des Anwendungsentwicklers
- ▶ Keine explizite Unterstützung für
  - ▶ Behandlung von Netzwerkpartitionierungen
  - ▶ bösartige Fehler
  - ▶ Softwarefehler und Designfehler
- ▶ Interoperabilität
  - ▶ Alle Replikate eines replizierten Objektes müssen die gleiche Infrastruktur nützen

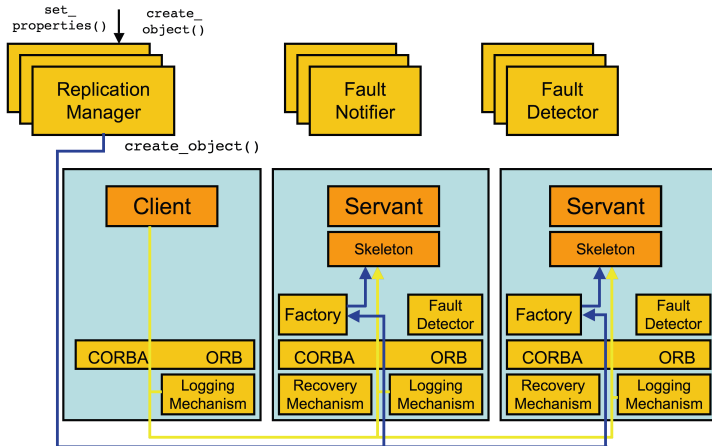
Es ist Aufgabe der Hersteller hier individuelle Lösungen anzubieten!

# Architekturübersicht



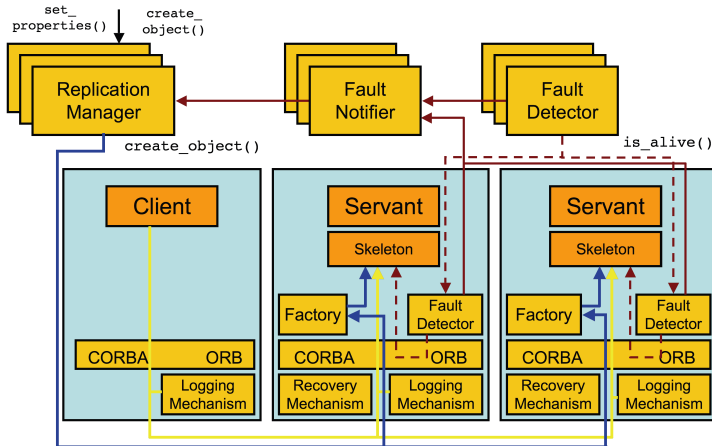


# Architekturübersicht





# Architekturübersicht



# Adressierung

## Interoperable Object Group Reference (IOGR)

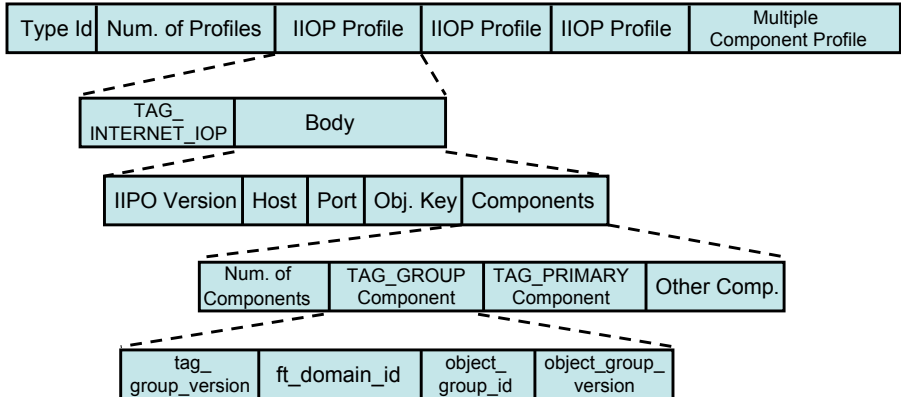
- ▶ Eine IOGR besitzt mehrere IOR Profile
- ▶ Jedes Profile setzt enthält Informationen zur Identität des Objektes
  - ▶ `FTDomainId` identifiziert die Domäne
  - ▶ `ObjectGroupId` identifiziert das replizierte Objekt (innerhalb der Domäne)
  - ▶ `ObjectGroupRefVersion` legt die Version der Referenz fest
- ▶ Maximal ein Profil enthält die Komponente `TAG_PRIMARY` zur Identifikation des primären Replikats.
  - ▶ Achtung, muss nicht aktuell sein!

## Was wird adressiert?

- ▶ direkte Adressierung der Replikate
- ▶ Adressierung von Gateways

# Adressierung

## Interoperable Object Group Reference (IOGR)



# Adressierung

## Aktualität von IOGRs

- ▶ Problem
  - ▶ Objektreferenzen können veralten, da sich die referenzierte Objektgruppe verändert
    - ▶ einzelne Replikate fallen aus, es kommen neue Replikate hinzu
- ▶ Lösung
  - ▶ Version der Client-Referenz wird bei Anfragen an den Server übertragen
    - ▶ Versionsinformation aus der Referenz wird als `GROUP_VERSION` Service Context der Anfrage beigefügt

# Adressierung

## Aktualität von IOGRs

- ▶ Lösung (Fortsetzung)
  - ▶ Server entnimmt Anfragen die Versionsinformation
  - ▶ Ist die Version des Clients aktuell wird die Anfrage verarbeitet
  - ▶ Ist die Version des Clients veraltet wird eine `LOCATE_FORWARD_PERM` Exception erzeugt
  - ▶ Ist die Version des Servers (anscheinend) veraltet wird der Replication Manager nach der aktuellen Referenz gefragt

# Verhalten bei Ausfällen

## Wiederholung von Aufrufen bei Ausfällen von Replikaten

- ▶ Auf ORB-Transportebene gibt es eine der folgenden Exceptions
  - ▶ `COMM_FAILURE`, `TRANSIENT`, `NO_RESPONSE`, `OBJ_ADAPTER`
  - ▶ Status ist `COMPLETED_MAYBE`
- ▶ Problem
  - ▶ Ohne weitere Maßnahmen kann eine Verletzung der at-most-once Semantik vorliegen
- ▶ Lösung
  - ▶ Eindeutige Identifizierung von Anfragen durch `REQUEST Service Context`
    - ▶ `Client Id`, identifiziert den Client
    - ▶ `Retention Id`, identifiziert den Aufruf
    - ▶ `Expiration Time`, legt fest wie lange Aufrufergebnisse aufbewahrt werden
  - ▶ Server ORB erkennt so die Wiederholungen von Anfragen und sendet das bereits ermittelte Ergebnis

# Verhalten bei Ausfällen

## ► Problem

- Es kommt zu einem Serverausfall oder Verbindungsproblem während eines Aufrufs
- Die TCP/IP Verbindung wird nicht ordnungsgemäß beendet und der Client bekommt den Abbruch deshalb nicht mit

## ► Lösung

- Periodische Testaufrufe (heartbeat messages)
- Client fragt diese via Policy pro Verbindung an und spezifiziert
  - Aufruffrequenz
  - Timeout
- Client-ORB ruft `_FT_HB()` beim Server-ORB auf
- Operation wird vom Skeleton bereitgestellt
- Server-ORB muss dies explizit erlauben
  - Kontrolle von erzeugtem Netzwerkverkehr

# Einstellung von Fehlertoleranzeigenschaften

## Konfigurierbare Eigenschaften

- ▶ Replikation
- ▶ Mitgliedschaft
- ▶ Konsistenz
- ▶ Monitoring
  - ▶ Intervall und Timeout
- ▶ Konfiguration von Factories
- ▶ Replikatanzahl
  - ▶ initiale Anzahl beim Start eines Dienstes
  - ▶ minimale Anzahl
- ▶ Sicherungspunktintervall



# Replikation

## Stateless

- ▶ statische Daten und nur lesender Zugriff

## Cold Passive Replication

- ▶ Recovery wird durch Sicherungspunkt und Protokollierung von Aufrufen erreicht

⇒ langsame Kompensation von Ausfällen

## Warm Passive Replication

- ▶ Aktueller Zustand des primären Replikats wird periodisch an alle anderen Replikate übertragen
- ▶ Protokollierung von Aufrufen

⇒ schnelleres Recovery im Vergleich zu cold passive

## Active Replication

- ▶ alle Replikate bearbeiten Aufrufe

⇒ sehr geringe Verzögerung bei Ausfällen

# Active Replication

## Normalbetrieb

- ▶ Wenn replizierte Objekte von anderen replizierten Objekten aufgerufen werden müssen duplizierte Aufrufe und Antworten unterdrückt werden

## Behandlung von Ausfällen

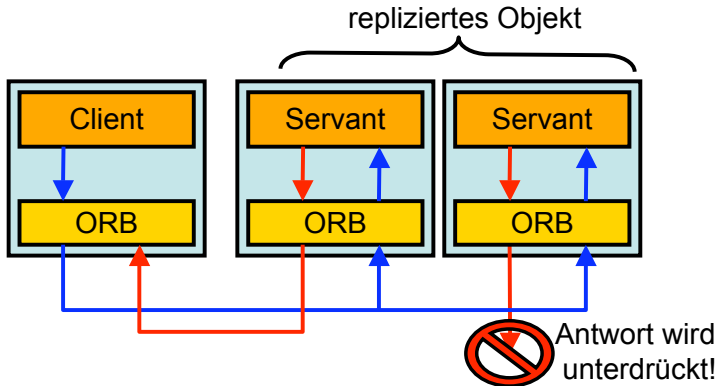
- ▶ Infrastruktur verschattet Ausfälle transparent da mindestens ein Replikat auf Anfragen antwortet

## Behandlung von Beitritten

- ▶ Zustandstransfer zur Integration eines neuen oder temporär ausgefallenen Replikats nötig

# Active Replication

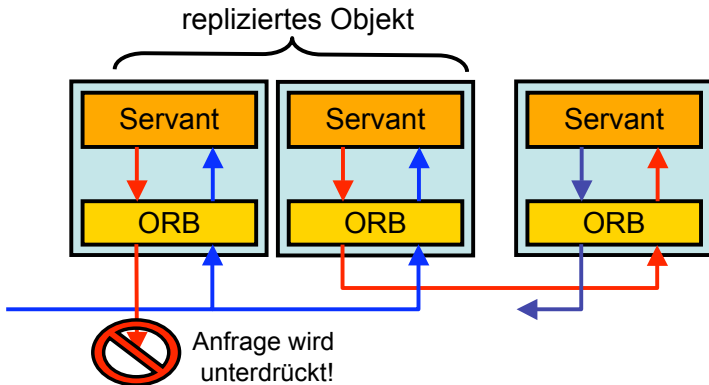
## Unterdrückung von duplizierten Ergebnissen



# Active Replication

## Unterdrückung von duplizierten Aufrufen

- Nötig wenn ein repliziertes Objekt andere Objekte aufruft



# Passive Replication

## Normalbetrieb

- ▶ periodischer Zustandstransfer zu allen Replikaten (nur bei warm passive nötig)
- ▶ Logging von Aufrufen und Sicherungspunkten

## Behandlung von Ausfällen

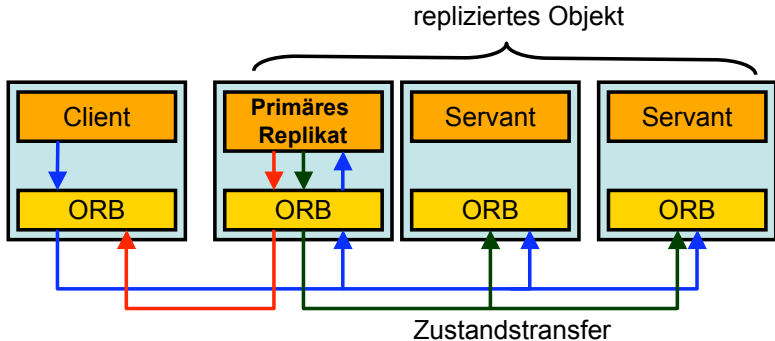
- ▶ Ausfall des primären Replikats erfordern die Wahl eines neuen Replikats
- ▶ Initialisierung des neuen primären Replikats (Zustandstransfer bei cold passive Replication)
- ▶ Erkennung von dublizieren Anfragen

## Behandlung von Beitritten

- ▶ Zustandstransfer zu neuem oder wiederhergestelltem Replikat bei warm passive Replication

# Passive Replication

## Unterdrückung von Anfragen und Zustandstransfer



# Vergleich der Replikationsformen

## Passive Replication

- ▶ Geringerer Ressourcenbedarf (z.B. Speicher und CPU)
- ▶ Langsame Erholung von Ausfällen

## Active Replication

- ▶ Hoher Ressourcenbedarf
  - ▶ Ausführung von Anfragen durch alle Replikaten
- ▶ Schnelle Kompensation von Ausfällen

Beide Ansätze erfordern die gleichen Mechanismen!

# Mitgliedschaft

## Infrastruktur kontrolliert Replikaterzeugung

- ▶ Infrastruktur erzeugt Replikate und platziert diese im Kontext der Domäne

## Anwendung kontrolliert Replikaterzeugung

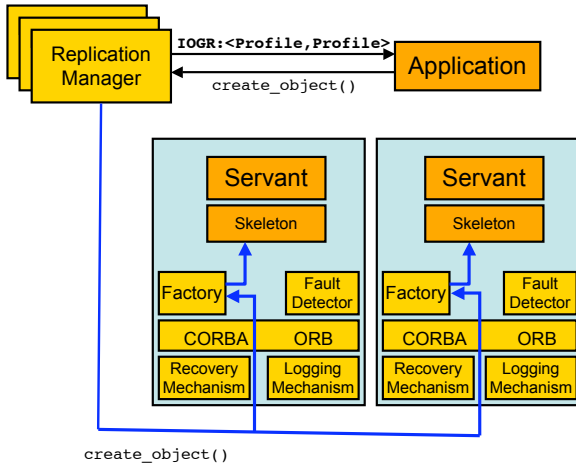
- ▶ Anwendung erzeugt und platziert Replikate entsprechend ihrer Erfordernisse



# Mitgliedschaft

## Infrastruktur kontrolliert Replikaterzeugung

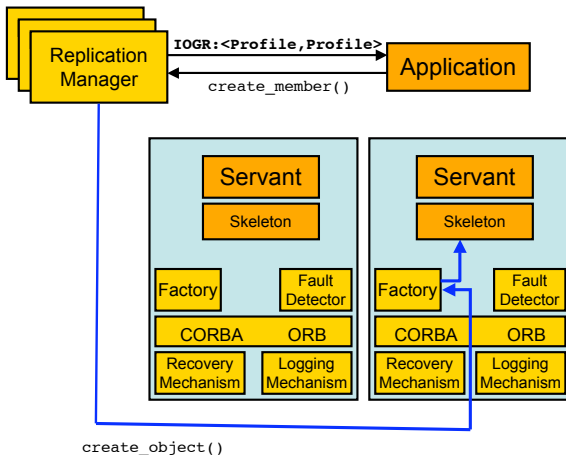
- Anwendung ruft `create_object()`-Methode des Replication Managers auf



# Mitgliedschaft

## Anwendung kontrolliert Replikaterzeugung

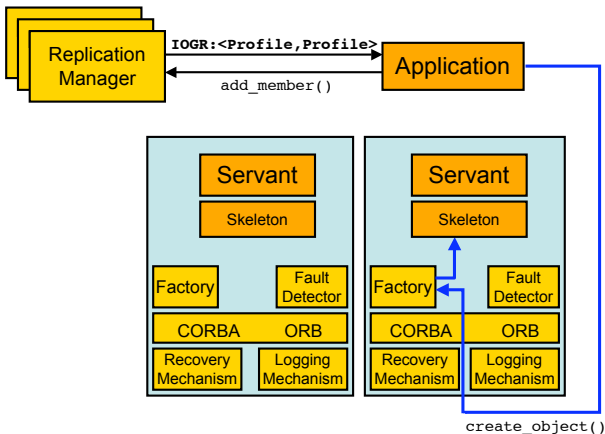
- Anwendung beauftragt den Replication Manager zur Erzeugung einzelner Replikate



# Mitgliedschaft

## Anwendung kontrolliert Replikaterzeugung

- Anwendung erzeugt Replikate eigenständig und nutzt den Replication Manager zur Verwaltung



# Konsistenz

## Infrastruktur kontrolliert Konsistenz

- ▶ Infrastruktur erhält die Konsistenz durch Mechanismen wie Logging, Zustandssicherung und Recovery
  - ▶ Aktive Replication
    - ▶ Nach jedem Aufruf haben alle Replikate den gleichen Zustand
    - ▶ Erfordert eine Gruppenkommunikation und total geordnete Zustellung von Nachrichten
  - ▶ Passive Replication
    - ▶ Nach jedem Zustandstransfer haben alle Replikate den gleichen Zustand

## Infrastruktur kontrolliert Konsistenz

- ▶ Anwendung stellt alle Mechanismen zur Konsistenzerhaltung bereit

# Monitoring

## Granularität der Beobachtung

### ► Mitglieder

- Es wird jedes Mitglied einer Objektgruppe beobachtet

### ► Ort

- Es wird ein Objekt als Repräsentant für den Ort beobachtet
- Wird das beobachtete Replikat beendet wird ein anderes Objekt ausgewählt
- Fällt das Objekt aus wird der Rechner als ausgefallen betrachtet

### ► Ort und Typ

- Es wird ein Objekt pro Ort und Typ kontrolliert.
- Wird das beobachtete Replikat beendet wird ein anderes Objekt vom gleichen Typ ausgewählt
- Fällt das Objekt aus werden alle Objekte des Typs an diesem Ort als ausgefallen betrachtet

# Factories

## Sequenz von FactoryInfo Strukturen

- ▶ Erzeugung von replizierten Objekten wird durch eine Sequenz von FactoryInfo Strukturen konfiguriert
- ▶ Aufbau der FactoryInfo Struktur
  - ▶ Referenz auf Fabrik
  - ▶ Ort der Fabrik
  - ▶ Information zur Konfiguration der Fabrik (Criteria)

# Verwaltung von Replikaten

## Aufgaben des Replication Manager

- ▶ Verwaltung von Objektgruppen und ihren Eigenschaften
  - ▶ Replikationsart
  - ▶ Mitgliedschaft
  - ▶ Konsistenz
  - ▶ usw.

# Schnittstelle des Replication Manager

- ▶ Methoden zur Benachrichtigung von Ausfällen
  - ▶ `register_fault_notifier()`
  - ▶ `get_fault_notifier()`
- ▶ Erbt von
  - ▶ Property Manager – Management von Fehlertoleranzeigenschaften
  - ▶ Object Group Manager – Verwaltung von Objektgruppen
  - ▶ Generic Factory – Erzeugung von replizierten Objekten



# Property Manager Schnittstelle

## Ermöglicht konfigurieren von Eigenschaften

- ▶ für alle Objektgruppen
- ▶ für alle replizierten Objekte eines bestimmten Typs
- ▶ für ein bestimmtes repliziertes Objekt zum Zeitpunkt der Erzeugung
- ▶ für ein bestimmtes repliziertes Objekt zu Laufzeit

Spezifischere Definitionen haben höhere Priorität

# Property Manager Schnittstelle

- ▶ Setzen/Ermitteln der Standardwerte für alle replizierten Objekte
  - ▶ `set_default_properties()`
  - ▶ `get_default_properties()`
  - ▶ `remove_default_properties()`
- ▶ Setzen/Ermitteln der Werte für alle replizierten Objekte eines Typs
  - ▶ `set_type_properties()`
  - ▶ `get_type_properties()`
  - ▶ `remove_type_properties()`
- ▶ Setzen/Ermitteln der Werte für ein bestimmtes repliziertes Objekt
  - ▶ `set_properties_dynamically()`
  - ▶ `get_properties()`

# Zeitpunkt der Konfiguration

Eigenschaften	Standard	Typ	Erzeugung	Dynamisch
Replikation	X	X	X	
Mitgliedschaft	X	X	X	
Konsistenz	X	X		
Monitoring	X	X		
Granularität	X	X	X	X
Fabriken		X	X	X
Initiale Replikatanzahl	X	X	X	
Minimale Replikatanzahl	X	X	X	X

# Generic Factory Schnittstelle

- ▶ Wird vom Replication Manager und von Fabriken implementiert zum Erzeugen von replizierten Objekten

```
typedef Object ObjectGroup;  
typedef any FactoryCreationId;  
  
Object create_object(  
    in Typeldtype_id ,  
    in Criteria the_criteria ,  
    out FactoryCreationIdfactory_creation_id)  
raises(NoFactory , ObjectNotCreated , InvalidCriteria ,  
        InvalidProperty , CannotMeetCriteria);  
  
void delete_object(  
    in FactoryCreationIdfactory_creation_id)  
raises(ObjectNotFound);
```

# Object Group Manager

- ▶ Operationen der Object Group Manager Schnittstelle:
  - ▶ `create_member()`
  - ▶ `add_member()`
  - ▶ `set_primary_member()`
  - ▶ `remove_member()`
  - ▶ `locations_of_members()`
  - ▶ `get_object_group_ref()`
  - ▶ `get_object_group_id()`
  - ▶ `get_member_ref()`

# Object Group Manager

- Wichtige Operation `create_member` und `add_member`

```
ObjectGroup create_member(  
    in ObjectGroup object_group ,  
    in Location the_location , in Typeld type_id ,  
    in Criteria the_criteria)  
raises(ObjectGroupNotFound , MemberAlreadyPresent ,  
        NoFactory , ObjectNotCreated , InvalidCriteria  
        , ... );
```

```
ObjectGroup add_member(  
    in ObjectGroup object_group ,  
    in Location the_location , in Object member)  
raises(ObjectGroupNotFound , MemberAlreadyPresent ,  
        ObjectNotAdded );
```

# Fehlermanagement

## Fault Detector

- ▶ Erkennt Fehler und erzeugt Fehlerberichte

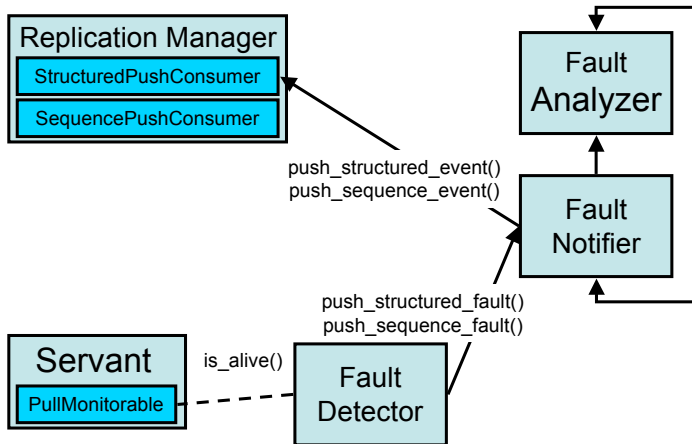
## Fault Notifier

- ▶ Sammelt und verarbeitet Fehlerberichte von Fault Detectors und Fault Analyzers
- ▶ Bildet eine Art Datenbank für alle Fehlerberichte

## Fault Analyzer

- ▶ Spezifisch für jede Anwendung
- ▶ Verarbeitet Fehlerberichte und fasst abhängige Fehler zusammen

# Fehlermanagement





# Fehlermanagement

## Fehlerweitergabe durch Fault Detectors

- ▶ einzelne Fehlerreporte (`CosNotification::StructuredEvent`)
- ▶ Sammelreport (`CosNotification::EventBatch`)
- ▶ Fehlertyp: (`ObjectCrashFault`)
  - ▶ `Domain_name` - FT\_CORBA
  - ▶ `Type_name` - `ObjectCrashFault`
  - ▶ `Location` - host/process
  - ▶ `TypeId` - IDL:Library:1.0
  - ▶ `ObjectGroupId` - 4711
- ▶ Sind alle Objekte eines Ortes ausgefallen wird `TypeId` und `ObjectGroupId` nicht im Fehlerreport vermerkt
- ▶ Sind alle Objekte eines Types ausgefallen wird die `ObjectGroupId` weggelassen

# Fehlermanagement

## Fehlerverarbeitung

- ▶ Fehlerberichte werden durch Fault Detectors erzeugt und via *push* an den Fault Notifier weitergeleitet
- ▶ Beliebige Konsumenten registrieren sich beim Fault Notifier und werden über Fehler benachrichtigt
  - ▶ Filter reduzieren das Nachrichtenaufkommen

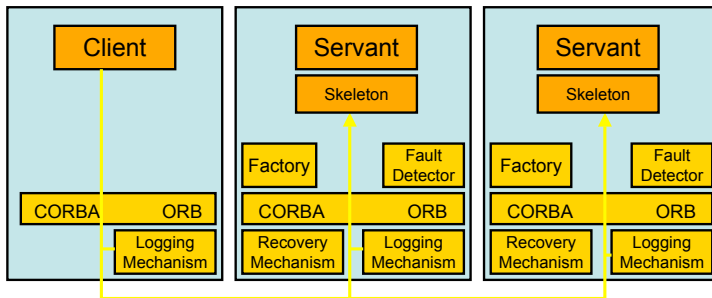
# Fehlermanagement

## Schnittstelle des Fault Notifier

```
void push_structured_fault(  
    in CosNotification::StructuredEvent event);  
  
CosNotifyFilter::Filter create_subscription_filter(  
    in string constraint_grammer)  
    raises(CosNotifyFilter::InvalidGrammer);  
  
ConsumerId connect_structured_fault_consumer(  
    in CosNotifyComm::StructuredPushConsumerconsumer ,  
    in CosNotifyFilter::Filter filter);
```

# Logging und Recovery

## Beispiel: Active Replication



# Logging und Recovery

## Schnittstelle zum Erzeugen Sicherungspunkten

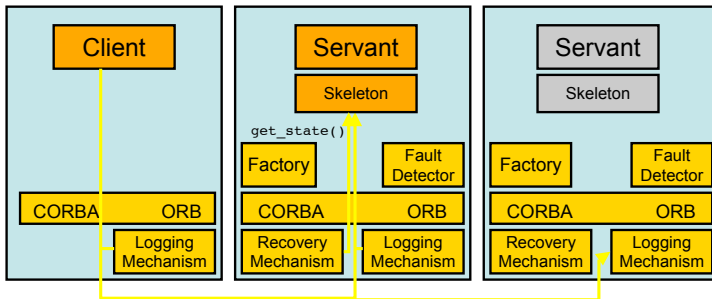
```
interface Checkpointable {  
    State get_state()  
        raises(NoStateAvailable);  
    void set_state(in State s)  
        raises(InvalidState);  
};
```

## Schnittstelle zum Erzeugen von partiellen Sicherungspunkten

```
interface Updateable : Checkpointable {  
    State get_update()  
        raises(NoUpdateAvailable);  
    void set_update(in State s)  
        raises(InvalidUpdate);  
};
```

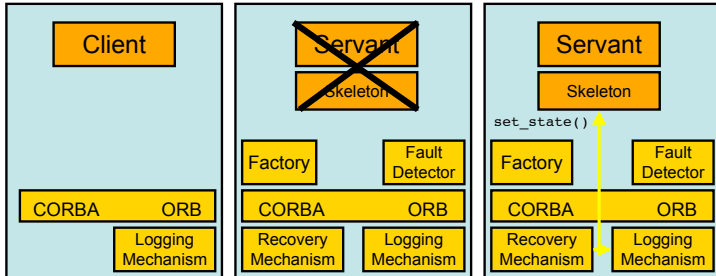
# Logging und Recovery

## Beispiel: Cold Passive Replication — Normalbetrieb



# Logging und Recovery

## Beispiel: Cold Passive Replication — Recovery



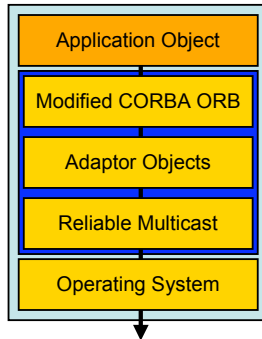
# Implementierungsvarianten

- ▶ Integrativer Ansatz
  - ▶ Unterstützung für replizierte Dienste wird direkt durch den ORB realisiert
- ▶ Dienst-Ansatz
  - ▶ Unterstützung für replizierte Dienste wird als CORBA Service angeboten
- ▶ Interceptor-Ansatz
  - ▶ Aufrufe werden durch Interceptoren (unterhalb des ORBs) abgefangen



# Integrativer Ansatz

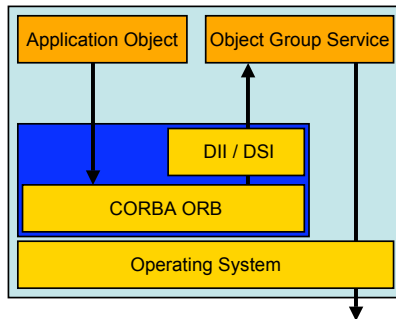
- ▶ ORB wird durch eine Gruppenkommunikation ergänzt
- ▶ IIOP wird durch ein proprietäres Protokoll ausgetauscht



- ▶ Nachteil: Aufwendige Implementierung
- ▶ Vorteil: Effizient und transparent für die Applikation und kann FT-CORBA kompatibel sein

# Dienst-Ansatz

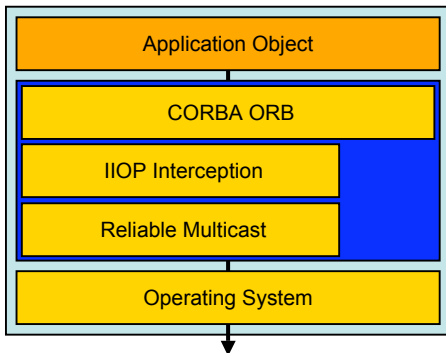
- Anwendung kommuniziert über lokale Dienste mit dem replizierten Objekt



- Nachteil: Indirektion über den ORB
- Nachteil: Nicht transparent für die Anwendung und nicht kompatibel
- Vorteil: Keine Veränderung des ORBs nötig und damit portable

# Interceptor-Ansatz

- ▶ Interceptoren fangen Aufrufe ab und vermitteln sie weiter an das replizierte Objekt



- ▶ Nachteil: Interceptoren sind oft spezifisch für ein Betriebssystem
- ▶ Vorteil: Transparent für die Anwendung

# Kommunikation

- ▶ Voraussetzung für aktive (und teilweise passive) Replikation ist in der Regel ein total geordneter Multicast (*abcast*)
  - ▶ Es gibt auf allen Nachrichten eine globale Reihenfolge
- ▶ Mögliche Realisierungen
  - ▶ Gruppenkommunikation nach dem *Virtual Synchrony Modell*
  - ▶ Einigungsalgorithmen (z.B. PAXOS von Lamport)

# Kommunikation

Beispiel: JGroups (<http://www.jgroups.org>)

- ▶ Java basierte Gruppenkommunikation
- ▶ realisiert Virtual Synchrony Modell
- ▶ Protokolle werden pro Applikation konfiguriert
- ▶ Beispielkonfiguration für aktive Replikation
  - ▶ TCP – Nachrichten werden via TCP übertragen
  - ▶ FD – Heartbeat Nachrichten zur Detektion von Ausfällen
  - ▶ UNICAST – Nachrichten an einzelne Mitglieder
  - ▶ NAKACK – Nachrichten an alle Mitglieder
  - ▶ STATE\_TRANSFER – Unterstützung für Zustandstransfer
  - ▶ GMS – Nachrichten zu Änderungen der Kommunikationsgruppe
  - ▶ STABLE – Garbage Collection von Nachrichten
  - ▶ TOTAL – Unterstützung für globale Reihenfolge von Nachrichten

# Kommunikation

## Beispiel: JGroups TOTAL

- ▶ Fester Knoten (Sequencer) bestimmt Nachrichtenreihenfolge
- ▶ Sequencer vergibt kontinuierliche Sequenznummer
  - ▶ UB: Sender an Sequencer, Sequencer an alle
  - ▶ BB: Sender an alle, Sequencer an alle
  - ▶ UUB: Sender an Sequencer, Sequencer an Sender, Sender an alle
- ▶ Empfänger puffert Nachrichten und liefert an Anwendung aus, falls `seqNr(erwartet) == seqNr(Nachricht)`
- ▶ Sequenzerausfall: View-Wechsel: Knoten auf gleichen Stand bzgl. alter View bringen, Neubeginn mit neuer View

Andere Beispiele: Spread, Ensemble, Rampart (byzantinisch)

# Zustandstransfer

## Vorraussetzungen

- ▶ Objekt muss die `Checkpointable`-Schnittstelle implementieren
- ▶ Erfordert den gesamten Objektzustand als `ByteArray` bereitzustellen
  - ▶ Aufgabe des Entwicklers!
- ▶ Sicherungspunkt kann erstellt werden wenn sich das Objekt in einem konsistenten Zustand befindet
  - ▶ alle laufenden Aufrufe werden beendet
  - ▶ neue Aufrufe werden blockiert

# Zustandstransfer

## Beispiel: Protokoll eines nicht-blockierenden Zustandstransfers

