Kapitel 6

TP-Monitore - Kommunikation

Inhalt

- Motivation
- **(T)RPC**
  - RPC
  - TRPC
  - Verknüpfung von RM zu Transaktionen
- Peer-to-Peer Messaging
- Queued Transaction Processing
- Zusammenfassung

Motivation

- Inter-Prozess-Kommunikation
  - Gründe
    - Verteilung
    - Heterogenität
    - modulare Auslegung der Komponenten
  - Möglichkeiten
    - (T)RPC: (Transactional) Remote Procedure Call
    - Peer-to-Peer Messaging
    - Queued Transaction Processing

Remote Procedure Call (1)

- RPC-Modell
  - Ziele
    - Inter-Prozess-Kommunikation möglichst genauso einfach gestalten wie der Prozeduraufruf innerhalb eines Prozesses
    - Verbergen von Heterogenität und dem eigentlichen Kommunikationsmechanismus vor dem Programmierer
    - Minimierung der Anzahl möglicher Fehlerarten
  - am weitesten verbreitet: OSF DCE (Open Software Foundation's Distributed Computing Environment)
  - Programmier-Modell
    - Beispiel

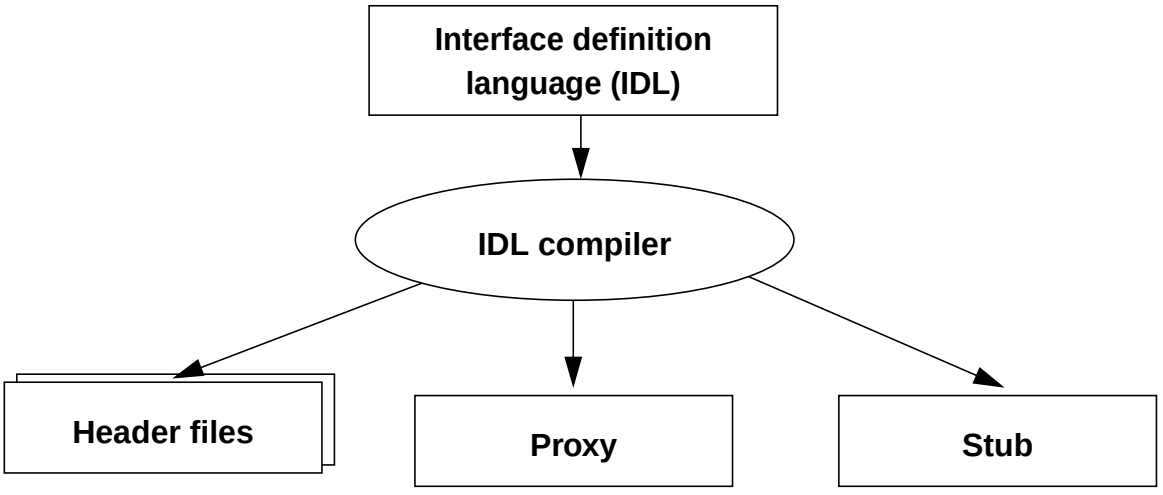
```
Boolean Procedure Pay_Bill (dda_acc#, cc_acct#)
{   int       dda_acct#, cc_acct#;
    long      cc_amount;
    Boolean   ok;
    Start;    /* start a transaction */
    cc_amount = pay_cc (cc_acct#);
    ok = debit_dda (dda_acct#, cc_amount);
    if (!ok) Abort; else Commit;
    return (ok);
}
long Procedure Pay_cc (acct#);
{   int       acct#;
    long      amt;
    /* get the credit card balance owed */
    Exec SQL  Select AMOUNT
              Into :amt
              From CREDIT_CARD
              Where (ACCT_NO = :acct#);
    /* set balance owed to zero */
    Exec SQL Update CREDIT_CARD
              Set AMOUNT = 0
              Where (ACCT_NO = :acct#);
    return (amt);
}
```

Remote Procedure Call (2)

- RPC-Modell (Forts.)
  - Programmier-Modell (Forts.)
    - Beispiel (Forts.)

```
Boolean Procedure Debit_dda (acct#, amt);
{   int       acct#,
    long      amt;

    /* debit amount from dda balance if balance is
sufficient */
    Exec SQL
        Update ACCOUNTS
        Set BALANCE = BALANCE - :amt
        Where (ACCT_NO = :acct# and BALANCE ≥ :amt);
    /* SQLCODE = 0 if previous statement succeeds */
    return (SQLCODE == 0);
}
```

  - Transaktionskontext wird in der Regel vom Aufrufer (Client) bestimmt
  - Client/Server-Kommunikation
    - Interface Definition Language (IDL)



Remote Procedure Call (3)

- RPC-Modell (Forts.)
  - Client/Server-Kommunikation (Forts.)

SQL-Precompiler

```
exec sql select A, B, C
         into     :v1, :v2, :v3
         from rel_a
         where D = :v4;
```

```
struct {...} format_CB;
struct {...} expression_CB;
struct {...} variable_CB;
...
expression_CB.params[1] = v4;
...
!sqlselect('fastsql',format_CB,
   expression_CB,&variable_CB);
...
v1 = variable_CB.result[1];
...
```

RM-Aufruf

Repository

rmType	Interface	Parameter Formats
sqlselect	caller.me	char(32)
sqlselect	caller.me	struct {...}
sqlselect	caller.me	struct {...}
sqlselect	caller.me	struct {...}
sqlselect	DBS@nodex	array of char
sqlselect	DBS@nodex	record of ...

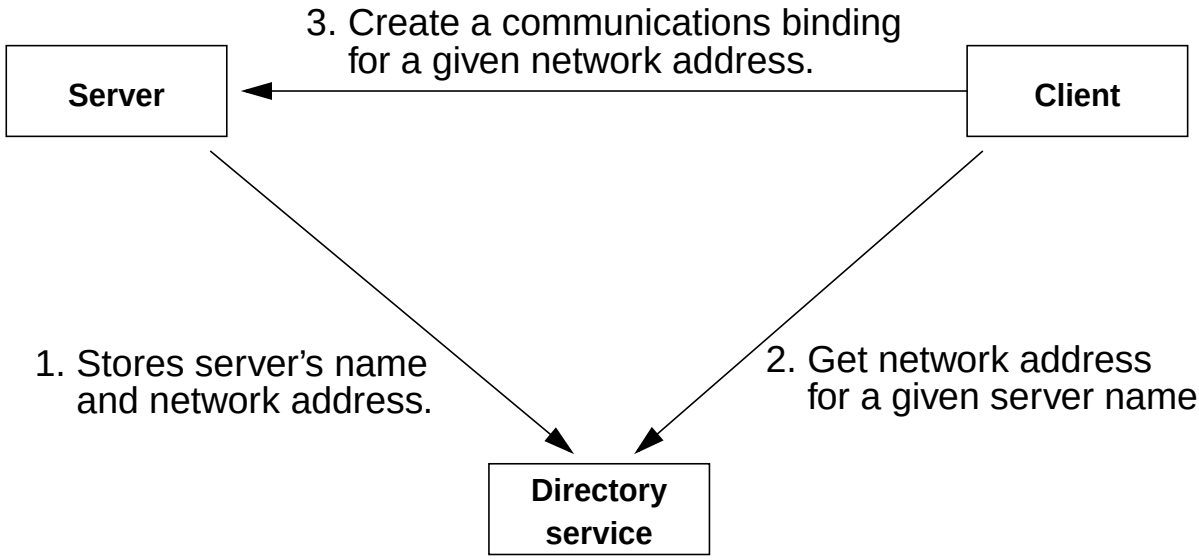
Stub-Compiler

RPC Stub

bind 'fastsql' to a RM ID
using a system RM
call (repository look-up)
coerce the caller's datatypes into the callee's datatypes, depending on who the RM call was bound to
pack all (converted) parameter values into a byte string that will serve as the message body for the RPC message
send off the request message(s)
wait
receive response message(s)
unpack the message body into the call parameters
return

Remote Procedure Call (4)

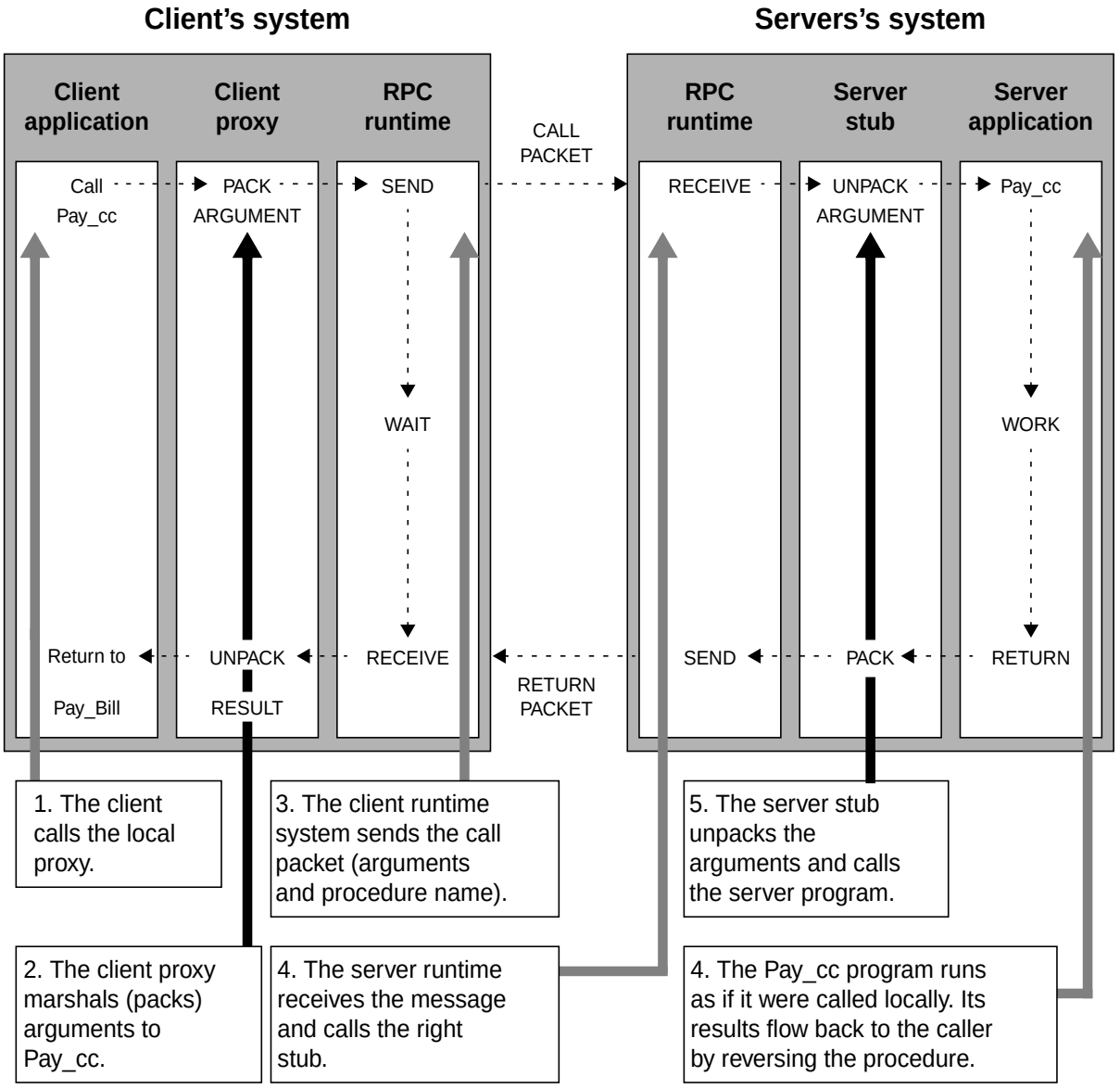
- RPC-Modell (Forts.)
  - Client/Server-Kommunikation (Forts.)
    - Marshalling
    - Directory Service



- man benötigt multi-threaded Client, um zu verhindern, dass Blockierung eines Aufrufers den Client-Prozess 'lahmlegt'
- Start-up-code zur Bindung der Programme vor der eigentlichen Kommunikation
  - Import/Export von Interfaces
  - Sicherheitseinstellungen
  - ...
- neue Exceptions zur Behandlung von Kommunikationsfehlern
  - z. B. Ergebnis kommt nicht

Remote Procedure Call (5)

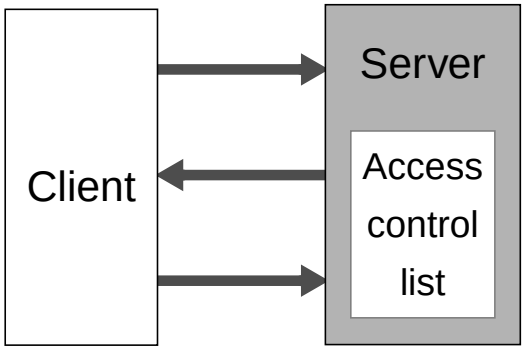
- RPC-Beispiel



Remote Procedure Call (6)

- Parameter-Behandlung
  - Möglichkeiten
    - Übersetzung in kanonisches Format
      - z.B. ISO-Standards ASN.1/BER und NDR oder Sun-Format XDR
      - insbesondere sinnvoll bei unterschiedlichen Programmiersprachen
    - *receiver-makes-it-right*: Server-Stub muss in der Lage sein, Client-Format zu interpretieren
      - effizienter
      - ausreichend bei gleichen Programmiersprachen und gleichem Speicherlayout der beteiligten Maschinen
      - Server verzichtet auf Transformation, wenn er erkennt, dass Parameter im 'eigenen' Format übertragen wurden
      - Server muss aber alle in Frage kommenden Formate erkennen und behandeln können

- Sicherheit



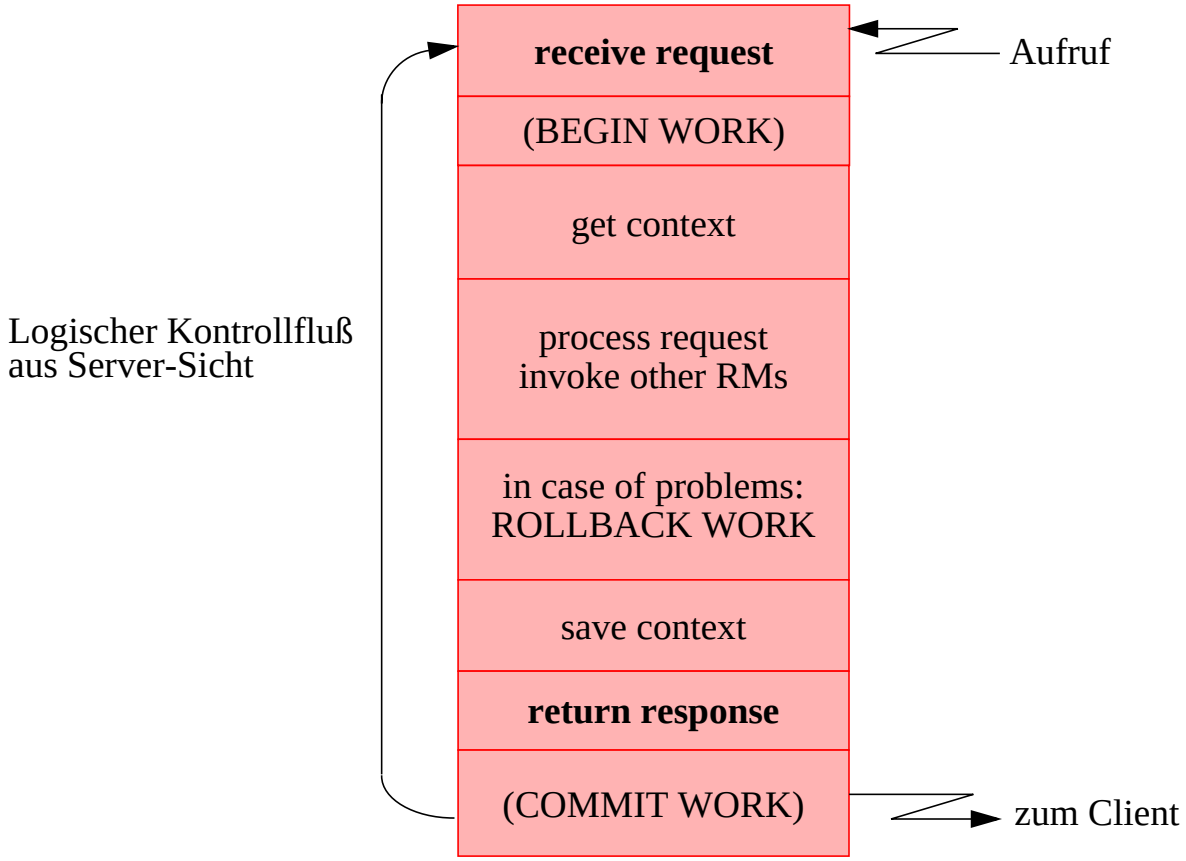
1. Create communicatons binding, passing in the authenticated client identity.
2. Acknowledge communications binding, passing back the authenticated server identity.
3. RPC to the server, which checks that the client is authorized to make this particular call.

Transactional Remote Procedure Call

- Fehlertoleranz
  - Ziel:
    - Ausführung eines idempotenten Servers ***mindestens einmal***
    - Ausführung eines nicht-idempotenten Servers (DB-Updates sind in der Regel nicht-idempotent) ***höchstens einmal***
  - Transaktionen helfen, '***genau einmal***' zu realisieren
- TRPC
  - Server sind RMs
    (Resource Manager, vgl. DTP-X/OPEN)
  - TRPC-Stub
    - wie RPC-Stub
    - zusätzlich jedoch als ***Agent des TP-Monitors*** verantwortlich für die TA-orientierte Kommunikation
  - Jeder RPC wird zum TRPC durch folgende TP-Monitor-Aktivitäten
    - Binden von RPC an Transaktionen durch ihre Kennzeichnung mit TRID
    - Informieren des TA-Mgr über den RM-Aufruf, wenn der TP-Monitor diesen verschickt
      (Registrieren der Teilnehmer an einer Transaktion)
    - Binden der Prozesse an Transaktionen:
      Im Fehlerfall (Crash) kann der TP-Monitor den TA-Mgr über den Ausfall des Prozesses informieren

Verknüpfung von RM zu Transaktionen (1)

❑ Grundstruktur einer transaktionalen Anwendung bzw. eines einfachen RM



➪ Ablauf

- Programm wartet oder wird vom TP-Monitor geladen/aktiviert
- es startet eine Transaktion, wenn der Auftrag noch keine TRID besitzt
- es verarbeitet den Auftrag
- falls das Server-Programm die äußerste SoC der Aktivität darstellt, wird COMMIT WORK aufgerufen

Verknüpfung von RM zu Transaktionen (2)

- Aufruf von RM
  - durch rmCall als generischen TRPC
  - mit Eingabe- und Ergebnisparametern in Byte-String (rmParams)

- Einfachste Form eines RM: IgnorantServer

```
Boolean IgnorantServer (rmParams *InParams, rmParams *OutResults)
  {       /*declares */
          work on input parameters                 /* example of the simplest version     */
          CompCode = rmCall('HelpMe', . . .);     /* of a transactional RM               */
                                                  /* checking of return code omitted     */
          prepare output parameters
          return(TRUE);
  }
```

Verknüpfung von RM zu Transaktionen (3)

- Einfachste Form eines RM: IgnorantServer (Forts.)
  - keine expliziten Vorkehrungen für Transaktionsschutz! falls keine wichtigen Berechnungen gemacht werden, gibt es nichts zu isolieren oder wiederherzustellen
  - falls *IgnorantServer* innerhalb einer TA aufgerufen wird, wird es jedoch automatisch Teil einer ACID-Einheit
  - auch wenn der Server *HelpMe* Transaktionsschutz braucht, muß *IgnorantServer* Teil einer TA sein, zumindest um die TRID an die aufgerufenen RM weiterzuleiten
  - das Programm kennt die TA-Verknüpfung nicht; sie liegt in der Verantwortung des TRPC-Mechanismus!
  - *IgnorantServer* wird, wie gezeigt, bei Commit nicht abstimmen
  - TA-Mgr kennt (über TP-Monitor) dieses Verhalten aus dem Katalog; deshalb wird für ihn immer ‘Commit’ angenommen, außer bei Crash des betreffenden Prozesses

Verknüpfung von RM zu Transaktionen (4)

- Einfache ACID-Transaktionen: SimpleServer

```
Boolean    SimpleServer (rmParams *InParams, rmParams *OutResults)
    {      TRID          NewTRID;                  /* TRID of the transaction started here   */
           RETCODE       CompCode;                  /* rmCall completion code                 */
           NewTRID = Begin_Work(. . .);             /* start transaction                      */
           do work;                                 /*                                        */
           CompCode = rmCall('HelpMe',. . .);       /* server invocation                      */
           if      (CompCode == BAD)                /* response OK?                           */
                   {       Abort_Work();            /* abort transaction                      */
                           return(FALSE);}          /* deny service                           */
           do more work;                            /*                                        */
           if      (result_ok)                      /*                                        */
                   {       prepare output parameters; /* example of an RM that              */
                           Commit_Work(. . .);      /* starts a TA and completes it           */
                                                    /* test for success of commit             */
                           return(TRUE);}           /* within the same invocation             */
           else    {       Abort_Work();            /*                                        */
                           return(FALSE); }         /*                                        */
    }                                               /*                                        */
```

Verknüpfung von RM zu Transaktionen (5)

- Einfache ACID-Transaktionen:
  SimpleServer (Forts.)
  - expliziter Beginn, Verarbeitung und
    in Abhängigkeit von Ergebnis: Commit oder Abort
  - RM läuft unter Transaktionsschutz ab, jedoch erst nach Zuordnung von TRID in NewTRID
  - Simple Server, Begin_Work, HelpMe u.a. nutzen den TRPC-Mechanismus

Verknüpfung von RM zu Transaktionen (6)

- Sicherungspunkte/Kontexte: CautiousServer
  - das Programm macht sich selbst schrittweise ‘recoverable’ durch Sicherungspunkte
  - Funktionsdefinitionen für CautiousServer

```
savepoint   Save_Work (Boolean persistent, context);
                /* take a savepoint with specified context; indicate if it has to be persistent.*/
savepoint   Rollback_Work (savepoint);
             /* return to the specified savepoint; returns the savepoint number actually reached*/
                                                /* might be prior to the one requested*/
context     Read_Context (savepoint);                          /* context from savepoint*/
TRID        Chain_Work (Boolean persistent, context);
              /* commit the ongoing transaction and pass its context to a new transaction; */
                                          /* this is only applicable to top-level transactions*/
```

  - Neuer Aspekt: Save_Work-Aufruf
    - kann interpretiert werden als spezieller Server, der Sicherungspunkte wartet
    - dieser Aufruf in einer TRPC-Umgebung wird automatisch an alle RMs weitergeleitet, die an der TA beteiligt sind
    - TA-Mgr veranlaßt HelpMe, sich auch am Sicherungspunkt zu beteiligen

Verknüpfung von RM zu Transaktionen (7)

- Sicherungspunkte/Kontexte: CautiousServer (Forts.)
  - Neuer Aspekt: Save_Work-Aufruf (Forts.)
    - umgekehrt wird bei Rollback_Work auf einen früheren Sicherungspunkt durch CautiousServer auch HelpMe gezwungen, sich auf diesen Sicherungspunkt zurückzusetzen
    - Nur Anwendungsprogramme, d. h. solche, die als Wurzel einer Aufrufhierarchie ablaufen, dürfen Sicherungspunkte erzeugen und auf solche zurücksetzen.
    - Was passiert sonst, wenn HelpMe Rollback_Work aufruft und anschließend die Kontrolle an CautiousServer zurückgibt?

Verknüpfung von RM zu Transaktionen (8)

- Sicherungspunkte/Kontexte: CautiousServer (Forts.)

```
Boolean      CautiousServer (rmParams *InParams, rmParams *OutResults)
  {   TRID          NewTRID;                    /* TRID of the transaction started here */
      savepoint     last_ok_state;              /* variable for remembered last savepoint */
      context       trusted_data;               /* storage area to be kept with savepoint */
      int           steps_p_savepoint;          /* how much work between savepoint */
      RETCODE       CompCode;                   /* rmCall completion code */
      initialize trusted_data
      NewTRID = Begin_Work(. . ., trusted_data, FALSE);
      last_ok_state = 1;                        /* begin is save_point no 1 */
      while (more_work)                         /* transaction goes on for a while */
      {     for (i=1; i<=steps_p_savepoint; i++)  /* in intervals call savepoint */
            {     do work;
                  CompCode = rmCall('HelpMe', . . . );   /* server invocation */
                  /* watch this space !!! */
                  do more work;
```

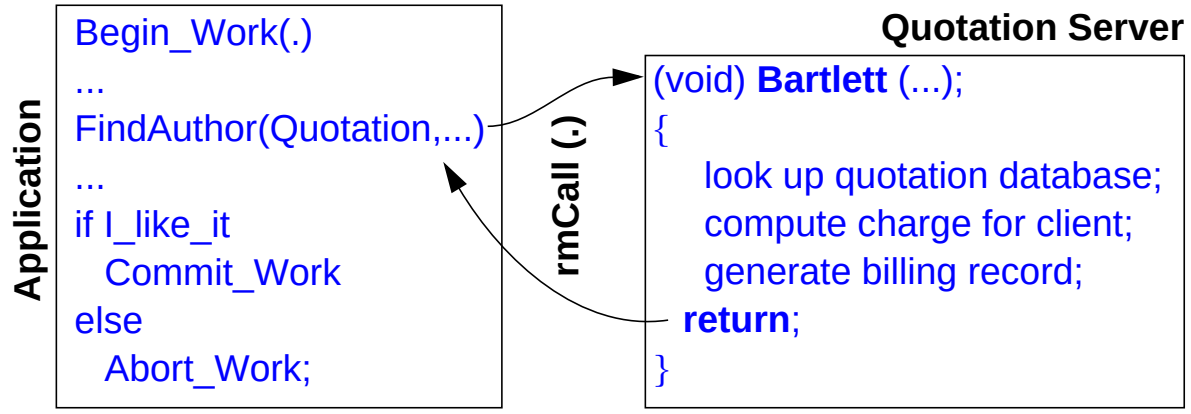
Verknüpfung von RM zu Transaktionen (9)

- Sicherungspunkte/Kontexte: CautiousServer (Forts.)

```
                if (error occurs)                          /* error: return to earlier savepoint */
                {       last_ok_state = Rollback_Work(last_ok_state);
                        trusted_data = Read_Context (last_ok_state);      /* read savepoint data */
                        reset local variables from what has been stored in trusted_data
                        i=0;                                    /* start new savepoint cycle */
        };      };                                                /* take a new savepoint */
        save all relevant local variables in trusted_data
        last_ok_state = Save_Work(FALSE, trusted_data); };
if      (result_ok)
{       prepare output parameters;                              /* example of an RM that */
        Commit_Work(. . .);                          /* uses savepoints to structure its work */
        return(TRUE);           }
else
{       Abort_Work();
        return(FALSE); }        }
```

Verknüpfung von RM zu Transaktionen (10)

- Umschalten zwischen Kontexten verschiedener TAs
  - **bisher:** RM liefen entweder innerhalb einer TA ab oder sie veranlassten den TA-Mgr, eine TA zu erzeugen (kontextfreie Verarbeitung)
  - **komplexere RM** "bedienen" mehrere TA gleichzeitig (multi-tasking etc.)
    - bei jedem Umschalten muß der richtige Kontext gewählt werden
    - Ablauf erfolgt nicht unter der TRID des Client
    - Rückkehr zum richtigen Client: TRPC muß eine dynamische Zuweisung und Umschaltung von Prozessen und Transaktionen ermöglichen
  - Einfaches Beispiel
    - Quotation-Server verlangt Gebühren, mit denen die rufende TA belastet werden soll - und zwar unabhängig vom Erfolg!
      - Erzeugen einer geschachtelten TA?
      - Kettung von TA?
      - Erzeugung einer unabhängigen TL-TA?



Verknüpfung von RM zu Transaktionen (11)

- Umschalten zwischen Kontexten verschiedener TAs (Forts.)
  - Schnittstelle zum TA-Mgr, um Arbeit für eine TA niederzulegen und wiederaufzunehmen

```
TRID      Leave_Transaction (void);
          /* tell the transaction manager that process is not working on current tranasaction */
          /* (for the moment); returns current TRID */

Boolean   Resume_Transaction (TRID desired);
          /* tells transaction manager that process wants to resume working on a transaction */
          /* it left earlier */
```

Verknüpfung von RM zu Transaktionen (12)

- Umschalten zwischen Kontexten verschiedener TAs (Forts.)

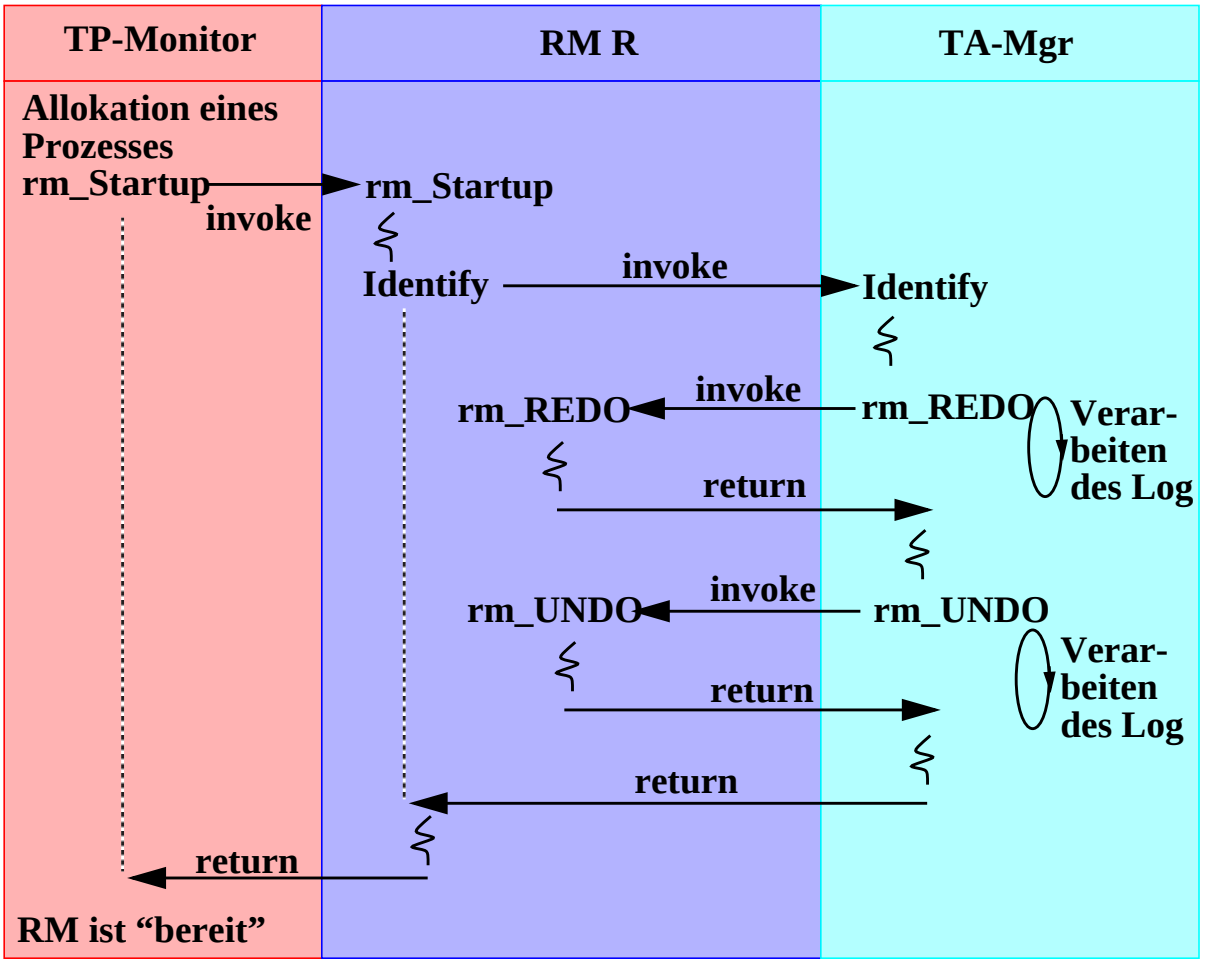
```
Boolean     Bartlett (rmParams *InParams, rmParams *OutParams)
                /* sample program for the dynamic creation of a nested top-level transaction */
{           ...
            TRID ClientTRID;                                    /* declare interface parameters */
            TRID MyownTRID;
            ...
            ClientTRID = Leave_Transaction();                   /* leave TA the client runs in */
            MyownTRID = Begin_Work(...);
                                    /* begin independent top-level TA, provide service requestetd */
            do database lookup
            prepare reference list
            write billing record to database                    /* now client will be charged */
            Commit_Work(FALSE, NULL);                           /* commit billing TA */
            Resume_Transaction(ClientTRID);   /* no error, the point is that server always commits */
            return;                                             /* return from server */ }
```

  - Server verhält sich genauso, als wenn er unter der TRID des Client ablaufen würde
  - Client wird durch eine Kostenrechnung belastet
  - Beim Commit der Anwendung verhält sich der Server so wie IgnorantServer

Verknüpfung von RM zu Transaktionen (13)

❑ Restart-Verarbeitung

➪ Basis-BS lädt Prozesse und eröffnet Kommunikationsleitungen; TP-Monitor besorgt das Laden der RM und die Überwachung ihrer Recovery



- RM wird geladen und identifiziert sich gegenüber TA-Mgr
- TA-Mgr überprüft den Log und gibt ggf. REDO/UNDO-Sätze an den RM, der die Recovery-Aktionen durchführt

Verknüpfung von RM zu Transaktionen (14)

- Nutzung persistenter Sicherungspunkte
  - Dauerhafte Konzepte
    - Wiederherstellbar nach einem Crash
    - einfache, anwendungsspezifische Lösung:
      - Speichere Kontext in DB
      - lies betreffendes Tupel nach Restart und setze Verarbeitung fort
    - keine Unterstützung durch TA-System: verteilte Lösung?

Verknüpfung von RM zu Transaktionen (15)

- Nutzung persistenter Sicherungspunkte (Forts.)
  - Beispiel: ComputeInterest

```
#define stepsize 1000;
#include <string.h>;
#define max_account_no 999999;
ComputeInterest (interest_rate)
    {       long account_no, last_account_done, batch_date;
            double account_total, interest_rate;
            int logsize;
            savepoint     HereWeAre;                    /* number of savepoint taken */
            context       FinishedAccount;              /* stored with the savepoint */
            last_account_done = 0; /* initialize the indicator of what has been completed so far */
            store last_account_done in FinishedAccount  /* prepare svpt context */
            Begin_Work(FinishedAccount, TRUE);
```

Verknüpfung von RM zu Transaktionen (16)

- Nutzung persistenter Sicherungspunkte (Forts.)
  - Beispiel: ComputeInterest (Forts.)

```
while (last_account_done < max_account_no)                    /*loop over accts.*/
    { exec sql UPDATE accounts
        SET account_total = account_total * (1+ :interest_rate)
        WHERE account_no BETWEEN
        :last_account_done + 1 AND :last_account_done + :stepsize;
        last_account_done = last_account_done + stepsize;
        store last_account_done in FinishedAccount              /* context */
        HereWeAre = Save_Work(FinishedAccount, TRUE);
                                          /* end of last savepoint interval */
Commit_Work( );                             /* commit whole transaction */
};
```

Verknüpfung von RM zu Transaktionen (17)

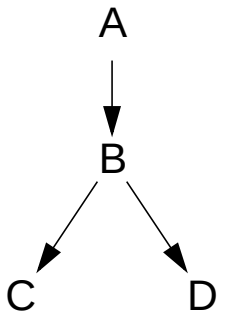
- Nutzung persistenter Sicherungspunkte (Forts.)
  - Rückkehr zu Sicherungspunkt
    - Flüchtiger Sicherungspunkt
      - Rollback_Work-Aufruf im Normalbetrieb
    - Persistenter Sicherungspunkt
      - im Gegensatz zu einem flüchtigen Sicherungspunkt kann ein persistenter auch bei Restart benutzt werden
      - jeder RM, der in der betreffenden TA teilnimmt, wird in den Zustand des letzten Sicherungspunktes versetzt
      - Laufzeitsystem der Programmiersprache muß RM sein, um den Zustand des Anwendungsprogramms im letzten Sicherungspunkt (lokale Variable, Aufruf-Stack, Befehlszähler, Heap, E/A-Ströme u.a.) wiederherzustellen
      - Anwendungsprogramm setzt die Verarbeitung so fort, als ob (nach außen) nichts geschehen wäre!

Peer-to-Peer Messaging (1)

- Rollen der Kommunikationspartner:
  - nicht *Master/Slave* (wie bei RPC),
  - sondern *Peers*
- Beispiel-Protokolle
  - IBM LU6.2 (dient im Folgenden als Beispiel)
  - ISO TP (siehe unten)
- Programmierschnittstelle
  - Beispiele für LU6.2
    - APPC (advanced program to program communications, dient im Folgenden als Beispiel, Kommandos korrespondieren stark mit den Protokollnachrichten von LU6.2)
    - CPI-C (common programming interface for communications, vereinfachte Version von APPC)
  - andere Beispiele
    - XATMI (Tuxedo)
    - sockets (TCP/IP)
    - TOP END communications manager
  - Konversationen
    - *Allocate*: Eröffnen einer Konversation
    - *Send_Data/Receive_Data*: Nachrichten versenden und empfangen
    - *Deallocate*: Beenden einer Konversation

Peer-to-Peer Messaging (2)

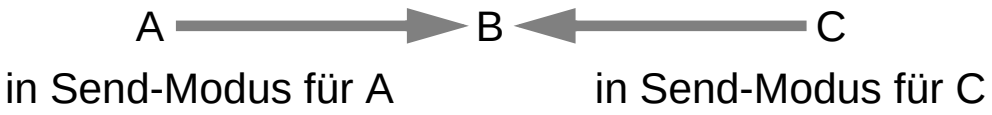
- Programmierschnittstelle (Forts.)
  - *Syncpoint*-Nachrichten für 2PC
    - APPC nutzt one-pipe-model (Synchronisations- und Anwendungsnachrichten in derselben Session)
  - *half dublex*
    - reflektiert Call/Return-Stil transaktionaler Kommunikation
    - zu jedem Zeitpunkt ist einer der Kommunikationspartner im *Send*, der andere im *Receive*-Modus
    - Rollen können mit *Receive_Data* gewechselt werden
  - Konversationsbäume
    - Programme eines Konversationsbaumes können an derselben verteilten Transaktion teilnehmen
    - bestimmen die Bedingungen, unter denen Commit ausgeführt werden kann, mit



  - Syncpoint-Regeln
    - Levels of synchronization: Koordinierung bei Commit
      - 0: keine Koordinierung
      - 1: applikations-spezifische Koordinierung
      - 2: Transaktionssemantik

Peer-to-Peer Messaging (3)

- Programmierschnittstelle (Forts.)
  - Syncpoint-Nachrichten
    - Nutzung, um den Nachbarn im Konversationsbaum anzuzeigen, dass Arbeit beendet werden kann
    - Voraussetzung: Senden einer Syncpoint-Nachricht erfordert entweder
      - dass alle Konversationen im Send-Modus sind und noch keine Syncpoint-Nachricht von einem anderen Programm eingegangen ist
      - alle bis auf eine der Konversationen sind im Send-Modus und über die sich im Receive-Modus befindliche Verbindung wurde bereits eine Syncpoint-Nachricht empfangen
    - diese Voraussetzungen können jedoch nicht alle Protokollfehler vermeiden



Peer-to-Peer Messaging (4)

- Beispiel

```
Boolean Procedure Pay_Bill (dda_acct#, cc_acct#)
{ int           dda_acct#, cc_acct#;
  long          cc_amount;
  Boolean       ok;
  CONVERSATION  conv_A, conv_B;
  /* Allocate the two conversations that are needed. */
  Allocate net_addr1.pay_cc Sync_level Syncpoint Returns conv_A;
  Allocate net_addr2. debit_dda Sync_level Syncpoint Returns conv_B;
  Send_data conv_A, cc_acct#;       /* Call Pay_cc */
  Receive_data conv_A, cc_amount;   /* get reply from Pay_cc */
  Receive_data conv_A, What_received=Send;/* put conv_A in Send mode */
  Send_data conv_B, cc_acc#, cc_amount;/* call Debit_dda */
  Syncpoint;                        /* start committing */
  Receive_data conv_B;              /* get Syncpoint from Debit_dda */
  If (What_received=Syncpoint) return (TRUE);
                                    /* committed */
  else return (FALSE);              /* aborted */
  Deallocate conv_A;                /* deallocate conversations */
  Deallocate conv_B;
}
```

Peer-to-Peer Messaging (5)

- Beispiel (Forts.)

```
void Procedure  Pay_cc (acct#);
{ int           acct#;
  long          amt;
  CONVERSATION  conv_C;

  Receive_allocate Returns conv_C;
  Receive_and_wait Gets acct#, Data_complete;
                                    /* get the call from Pay_Bill */
  Exec SQL      Select AMOUNT into :amt/* retrieve the amount owed */
                From CREDIT_CARD
                Where (ACCT_NO = acct#);
  Exec SQL      Update CREDIT_CARD /* pay the bill */
                Set AMOUNT = 0
                Where (ACCT_NO = acct#);
  Receive_and_wait What_received=Send;/* put conv_A into Send mode */
  Send_data amt;                    /* send reply to Pay_Bill */
  Receive_data What_received=Take_Syncpoint;
                                    /* get Syncpoint from Pay_Bill */
  Syncpoint;
}
```

Peer-to-Peer Messaging (6)

- Beispiel (Forts.)

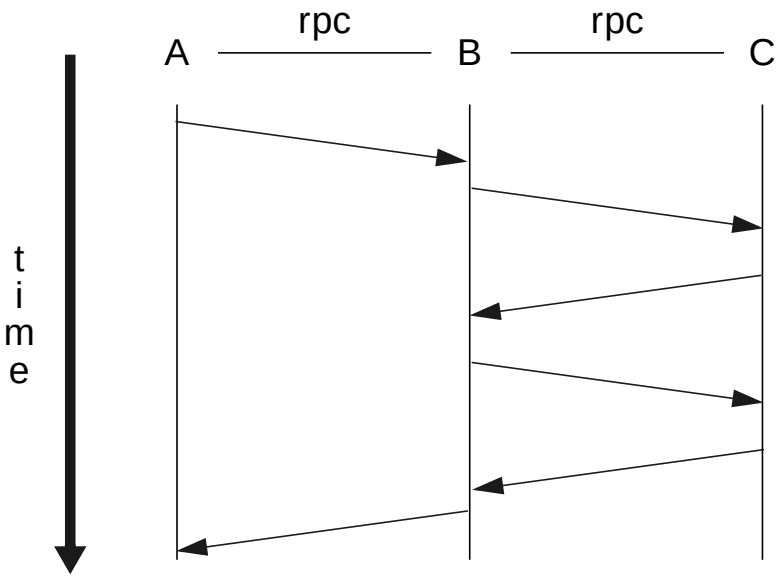
```
void Procedure     Debit_dda (acct#, amt);
{ int              acct#;
  long             amt;
  CONVERSATION     conv_C;

  Receive_allocate Returns conv_C;
  Receive_and_wait Gets acct#, amt Data_complete;
                                   /* get the call from Pay_Bill */
  Receive_and_wait What_received=Take_Syncpoint;
                                   /* get Syncpoint from Pay_Bill */
  Exec SQL      Update ACCOUNTS    /* debit the account by amt */
                Set BALANCE = BALANCE - :amt
                Where (ACCT_NO = :acct# and BALANCE ≥ amt);
  If (SQLCODE == 0) Syncpoint else Rollback;
}
```

Peer-to-Peer Messaging (7)

- ISO TP
  - global eindeutige TrIDs
  - nur die Wurzel eines Konversationsbaumes kann Commit anstoßen
  - Applikationen haben die Wahl zwischen *half*- und *full-dublex*
  - erlaubt auch *unchained*-Modus neben *chained*-Modus
  - ähnlich wie die X/Open DTP kann auch ISO TP auf RPC- oder Peer-to-Peer-Implementierungen aufsetzen und diese für den eigentlichen Datentransport nutzen

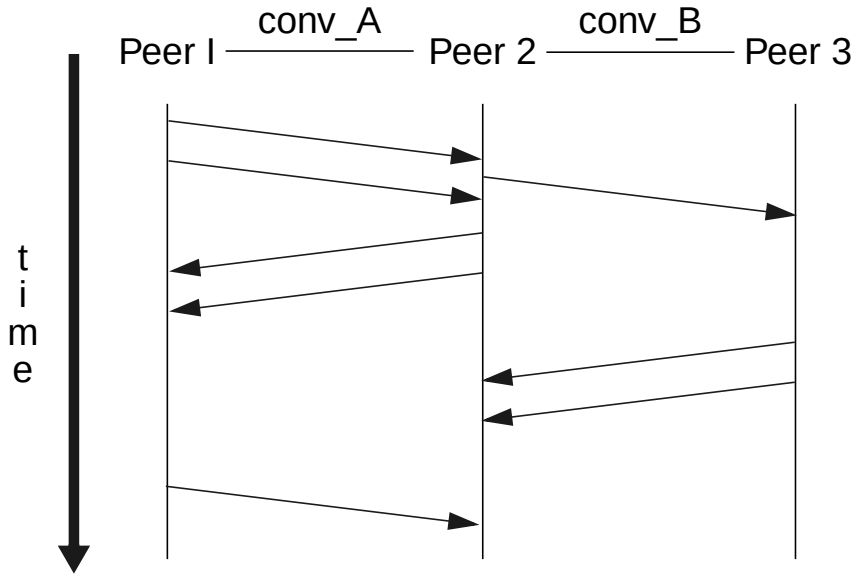
Peer-to-Peer Messaging (8)

- Vergleich mit RPC
  - Unterschiede
    - Flexibilität bzgl. der möglichen Nachrichtenfolgen
    - Möglichkeiten des Beendens von Transaktionen durch die beteiligten Programme
    - Verwaltung der Zustände verteilter Transaktionen
    - Auswirkungen auf die Programmierung
  - Nachrichtenfolgen
    - RPC: Call-Return / Schachtelung



Peer-to-Peer Messaging (9)

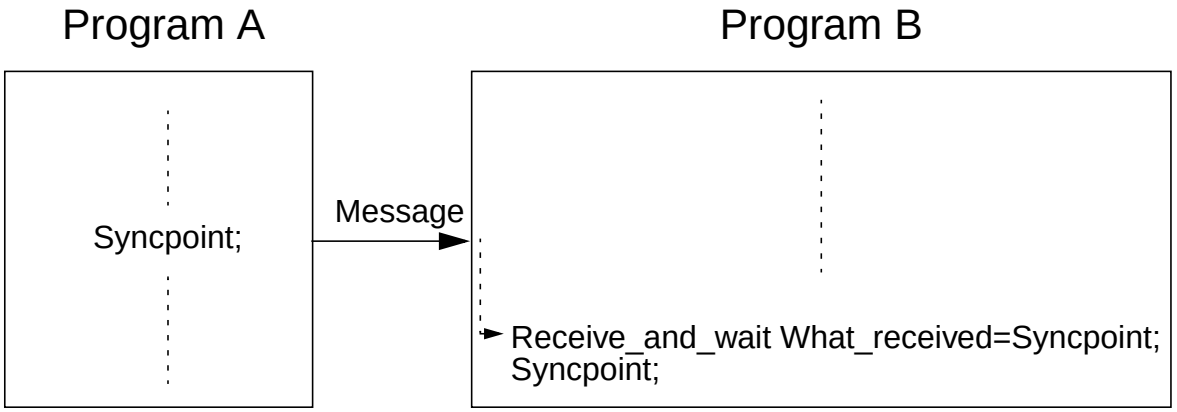
- Vergleich mit RPC (Forts.)
  - Nachrichtenfolgen (Forts.)
    - Peer-to-Peer
      - flexibler als bei RPC
      - Senden/Empfangen von Nachrichten jederzeit möglich (in LU6.2 im Rahmen von *half-dublex*)
      - Nachrichten müssen nicht entweder *Calls* oder *Returns* sein und müssen nicht dementsprechend geschachtelt sein
      - Nachrichtenfolgen möglich, die in RPC nicht auftreten könnten



      - Vorteile:
        höhere Flexibilität;
        mehr Parallelität möglich;
        kein Zwang, auf Antworten zu warten;
        Batching mehrerer Calls in einer Nachricht möglich;
      - Nachteil:
        erhöhte Komplexität in der Programmierung

Peer-to-Peer Messaging (10)

- Vergleich mit RPC (Forts.)
  - Terminierung
    - RPC
      - *termination by return to caller*
      - Beachte: Terminierung ≠ Commit
    - Peer-to-Peer
      - Ankündigung der Terminierung durch *Syncpoint*
      - durch *Syncpoint* kann Commit der Transaktion durch ein beliebiges der beteiligten Programme eingeleitet werden (auch bevor andere terminieren)
      - Verzögerung des Commit, wenn noch Arbeit zu verrichten



      - Vorteile:
        höhere Flexibilität;
        mehr Parallelität
      - Nachteile:
        Fehleranfälligkeit (z. B. Deadlock, wenn Programm1 Syncpoint aufruft und Programm2 auf Nachricht von Programm1 wartet),

Peer-to-Peer Messaging (11)

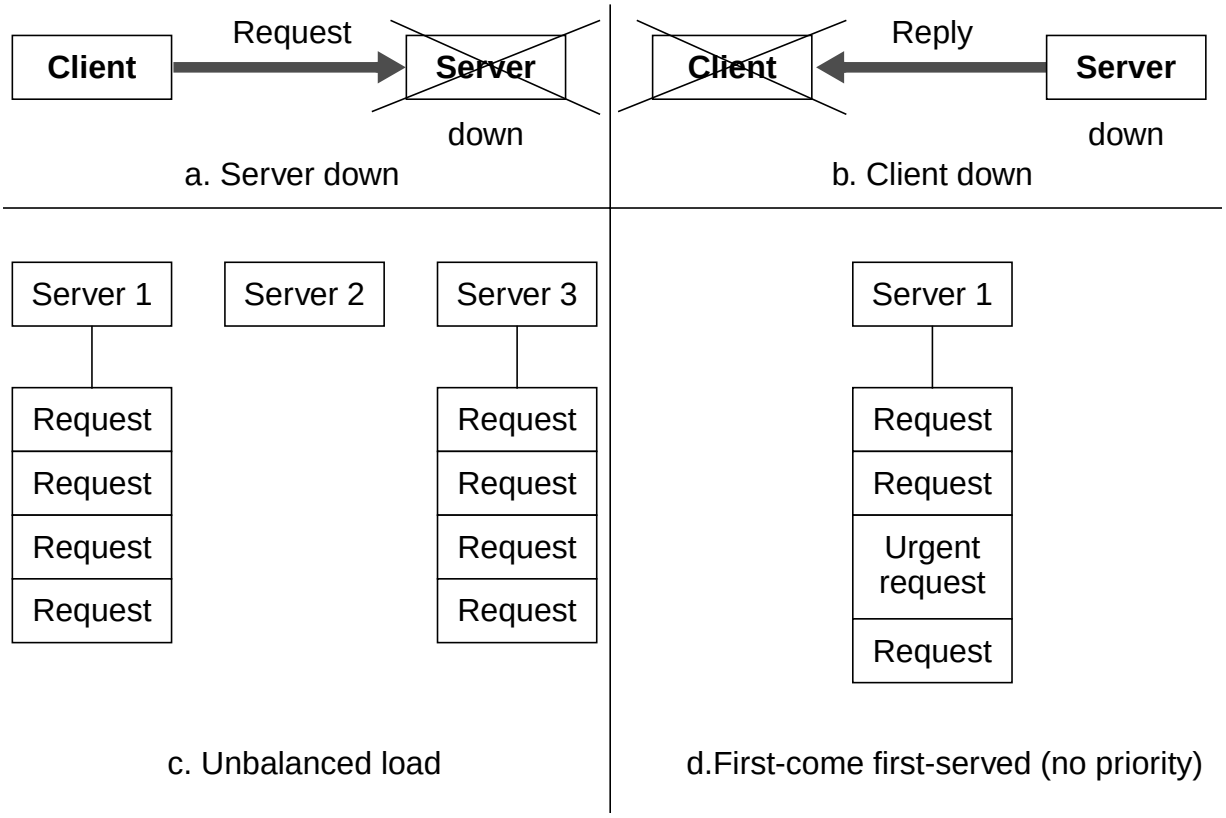
- Vergleich mit RPC (Forts.)
  - Verbindungsmodell
    - Peer-to-Peer
      - *conncection-oriented communication model*
      - gemeinsamer Zustand einer Konversation:
        Richtung der Kommunikation;
        Richtung des Links (*Send*- oder *Receive*-Modus);
        aktueller TrID;
        Zustand der Transaktion: *active*, *committed* oder *aborted*
      - Programm kann davon ausgehen, dass Kommunikationspartner den Zustand zum Zeitpunkt seiner letzten Nachricht beibehält
      - *Peer-to-Peer Connections* sind nicht *recoverable*; nach Fehler ist Programm für die Wiederherstellung des aktuellen Zustands verantwortlich
    - RPC
      - *conncectionless communication model*
      - kein gemeinsamer Zustand für Client und Server
      - viele RPC-Systeme bieten Unterstützung in Form eines *context handles* (z. B. DCE und MS RPC, der bei jedem Aufruf vom Client an den Server mitgeschickt wird)

Peer-to-Peer Messaging (12)

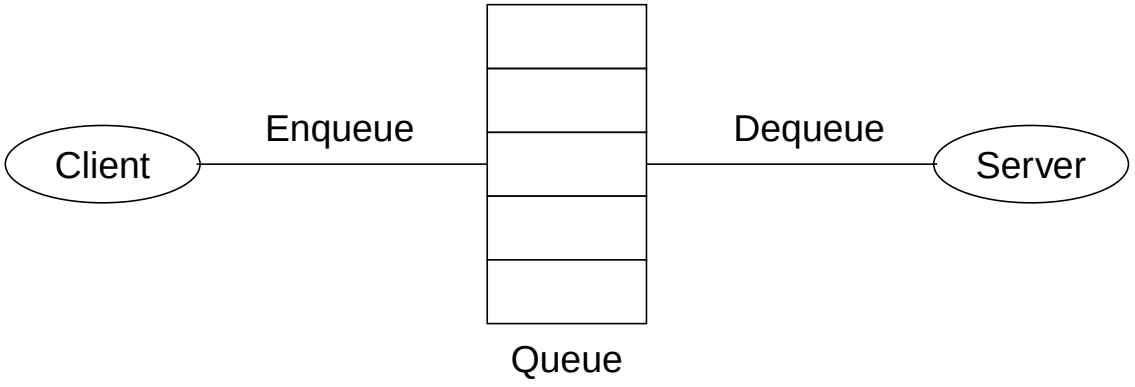
- Vergleich mit RPC (Forts.)
  - Aktuelle Situation
    - RPC einfacher zu nutzen und weit verbreitet
    - die meisten TP-Monitore/DBMS nutzen Peer-to-Peer: z. B.:
      - TDS over TCP/IT (MS und Sybase)
      - GCA over TCP/IP (Ingres)
      - DRDA over LU6.2 and TCP/IP (IBM)
      - TUXEDO (BEA)
      - TOP END (AT&T/NCR)
    - RPC-basiert:
      - SQL*Net (Oracle)
      - ACMS (Digital)
      - Pathway/TS (Tandem)

Queued Transaction Processing (1)

- Probleme synchroner Kommunikation

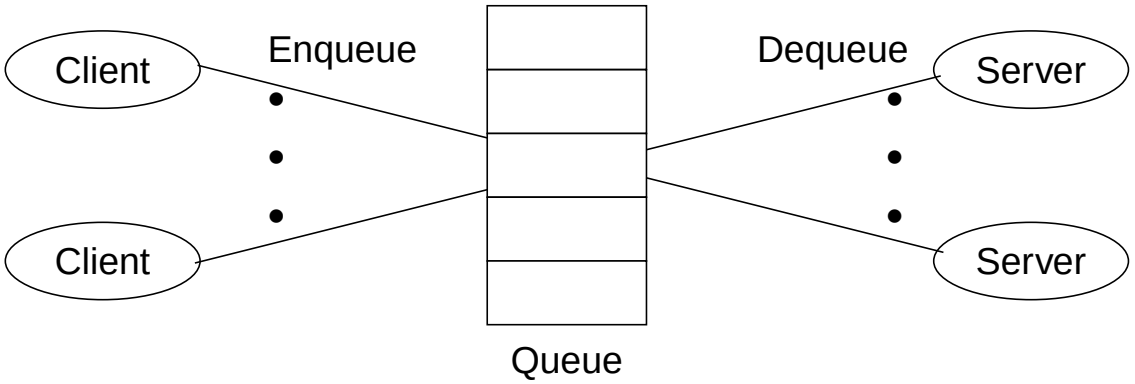


- Lösung: Queuing
  - Nutzung von Persistenten Warteschlangen

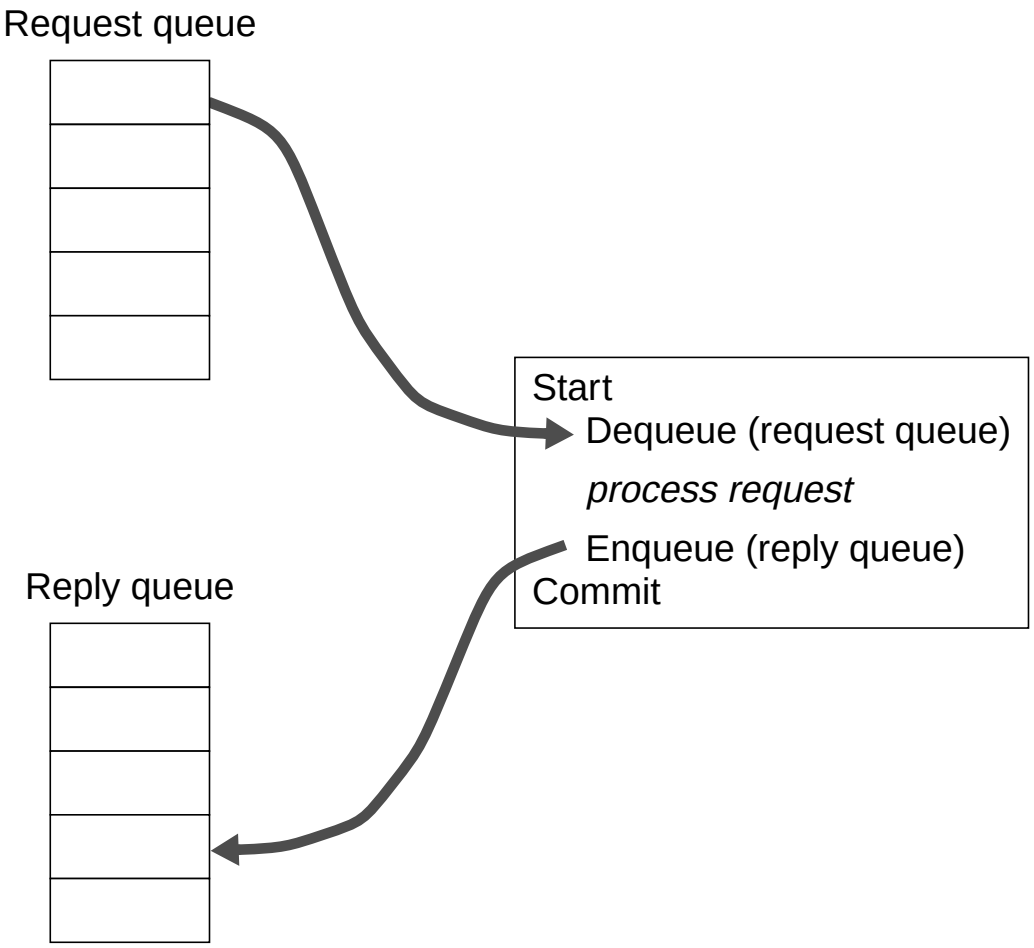


Queued Transaction Processing (2)

- Lastbalancierung

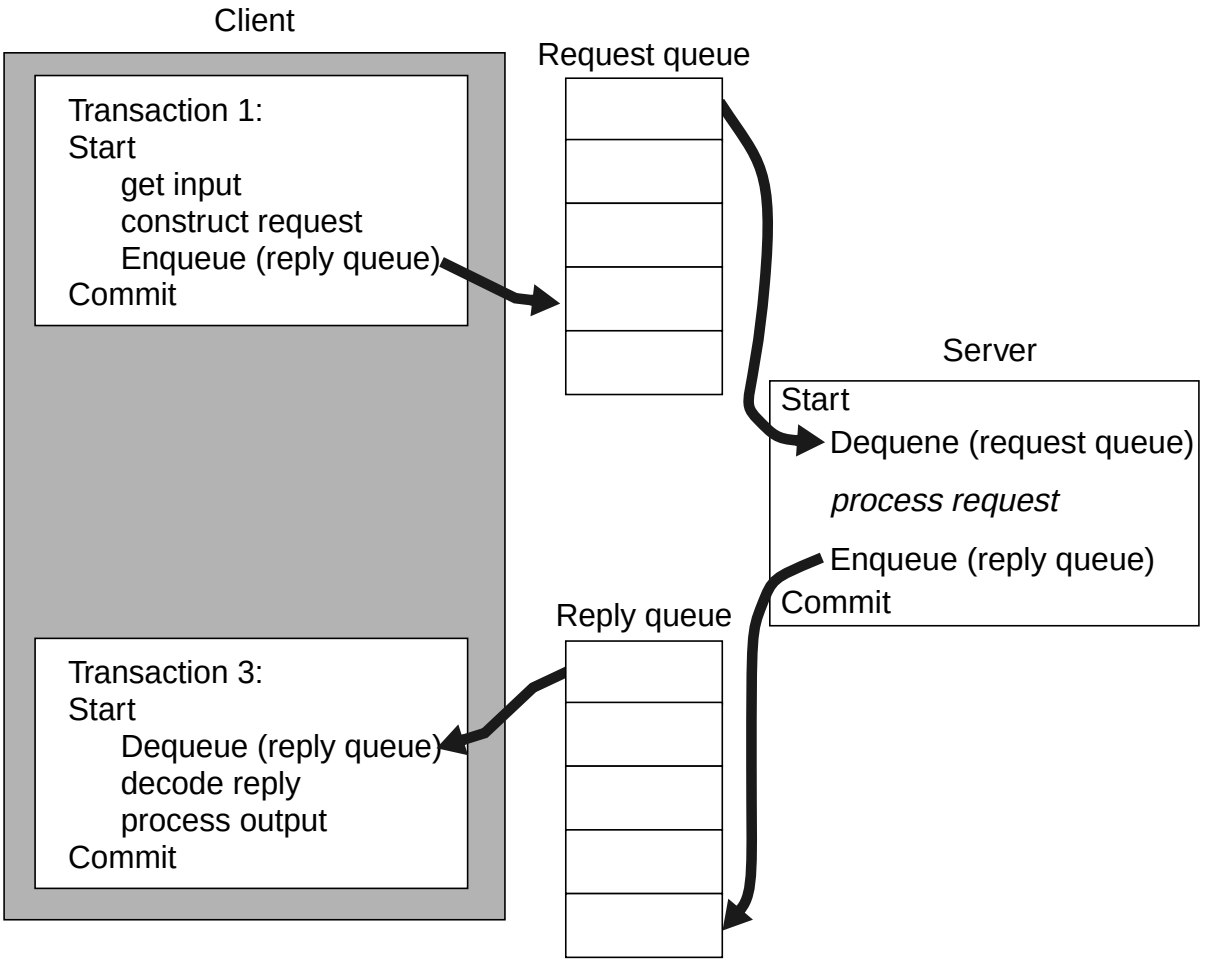


- Queues und Transaktionen



Queued Transaction Processing (3)

- Queues und Transaktionen (Forts.)



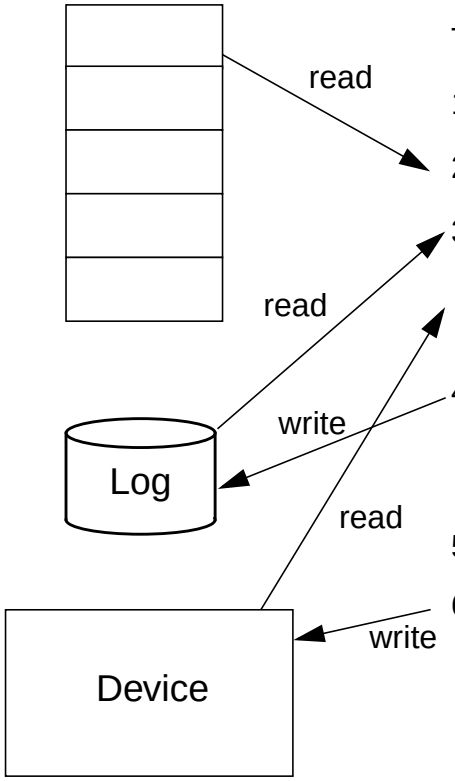
- Client Recovery
  - mögliche Zustände:
    1. kein Commit von T1
    2. Commit von T1, kein Commit von T2
    3. Commit von T2, kein Commit von T3
    4. Commit von T3

Queued Transaction Processing (4)

- Client Recovery (Forts.)
  - Vorgehensweise:
    - Clients versehen ihre Aufträge mit global eindeutigen Request-Identifikatoren (RIDs) und speichern diese selbst persistent
    - Queue-Manager hält für jeden Client die RIDs des letzen Requests, der eingelagert (Enqueue) wurde, und der letzen Antwort, die ausgelagert (Dequeue) wurde
    - Queue-Manager verwaltet Zustand der Session
    - Zur Recovery erfragt der Client die zugehörigen RIDs vom Queue-Manager
      - wenn RIDs ‘letzter Request’ ungleich: Zustand 1;
        Reaktion: Re-Submit oder neuer Request
      - wenn RIDs ‘letzte Antwort’ gleich: Zustand 4;
        Reaktion: neuer Request
      - RIDs ‘letzter Request’ gleich und RIDs ‘letzte Antwort’ ungleich und Reply Queue ist nicht-leer (Ann.: *one request at a time* und eigene private *Reply Queue*):
        Zustand 3;
        Reaktion: Dequeue und Starten von T3
      - RIDs ‘letzter Request’ gleich und RIDs ‘letzte Antwort’ ungleich und *Reply Queue* ist leer: Zustand 2
        Reaktion: auf Antwort warten

Queued Transaction Processing (5)

- Client Recovery (Forts.)
  - *Non-Undoable Operations* und System in Zustand 3



To process a reply

1. Start a transaction
2. Dequeue the reply.
3. If there is an earlier logged device state for this reply and it differs from the the current device state, then ask the operator whether to abort this transaction.
4. Log the current device state on persistent storage along with the reply's ID. This operation must be installed whether or not the transaction commits.
5. Perform the operation on the physical device.
6. Commit.

Queued Transaction Processing (6)

- Einsatz
  - in TP-Monitoren, bspw:
    - TUXEDO (BEA)
    - Encina (Transarc)
    - TOP END (AT&T/NCR)
  - als eigenständige Produkte, bspw:
    - DECmessageQ (Digital)
    - MQSeries (IBM)

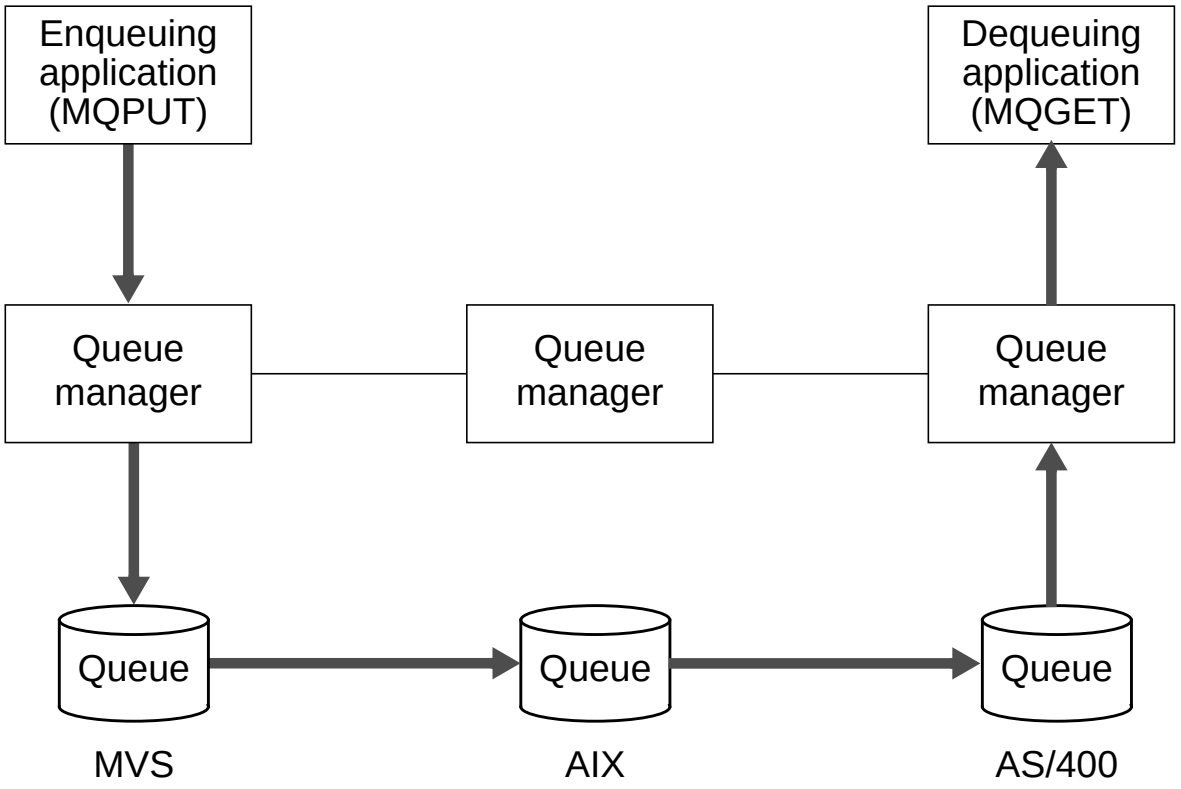
- Beispiel: MQSeries
  - von IBM als ‘Interoperabilitätslösung zwischen verschiedenen Betriebssystemen und TP-Umgebungen’ angeboten
  - API MQI (Message Queue Interface) kann von Applikationen genutzt werden, die
    - unter den TP-Monitoren von IBM, CICS und IMS laufen
    - und auf diversen BS (IBM MVS/ESA, OS/400, AIX/6000, OS/2 sowie AT&T/NCR UNIX, HP-UX, SCO UNIX, SunOS, Sun Solaris, Tandem NonStop Kernel, UnixWare, VAX/VMS, MS Windows NT)
  - Basisoperationen: MQPUT (Enqueue), MQGET (Dequeue)
  - mit Namen versehene Queue kann mehrere nebenläufige *Enqueuer* und *Dequeuer* bedienen

Queued Transaction Processing (6)

- Beispiel: MQSeries (Forts.)
  - MQPUT
    - Queue-Manager startet TA, die in die Applikations-TA mit-einbezogen werden kann, aber nicht muss
    - eingelagerte Nachricht:
      - Applikationsdaten
      - und *Message Context:*
        system-generierte Message-ID,
        von der Applikation zugewiesene Message-ID,
        Persistenz-Flag,
        Priorität,
        Name der *Destination Queue*,
        ggf. ID der *Reply Queue*,
        Message Type,
        Zuordnung von Request und Antwort,
        Expiration Time,
        applikations-spezifische Format-Information,
        Report Options,
        ...
  - Transaktionen
    - MQI-Operationen nehmen an Transaktionen teil, wenn die Applikationen unter Kontrolle eines TP-Monitors/TA-Managers laufen, der XA von DTP-X/Open unterstützt
    - wenn Applikation außerhalb einer Transaktion läuft, nutzt der Queue-Manager eigenen TA-Manager, der es erlaubt, mehrere MQI-Operationen zusammzufassen

Queued Transaction Processing (7)

- Beispiel: MQSeries (Forts.)
  - persistente und nicht-persistente Nachrichten können in derselben Queue verwaltet werden (nur bei MQSeries)
    - persistent: *exactly once*
    - nicht-persistent: *at most once*
  - Queue Forwarding
    - asynchrone und transaktionale Übertragung

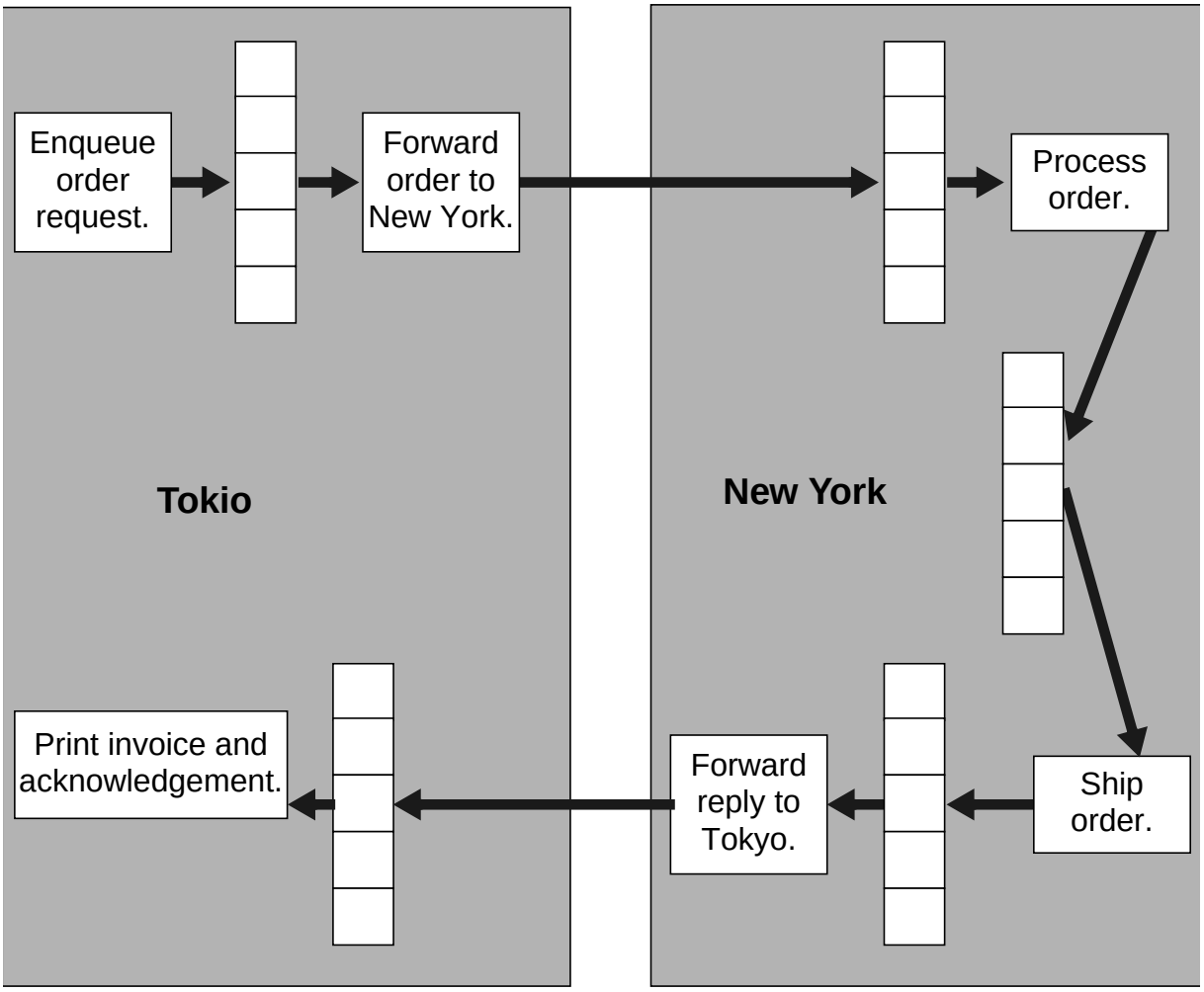


Queued Transaction Processing (8)

- Beispiel: MQSeries (Forts.)
  - MQGET
    - Einbeziehung in Applikations-TA
    - Blocking Option: wenn Queue leer, kann Aufrufer blockiert werden, bis Nachricht vorhanden oder Timeout
    - Auslesen der Nachrichten: *ordered* oder über ID/Key
  - Queue-Manager
    - kann mehrere Queues verwalten
    - Komponenten
      - *Connection Manager*: Verwaltung der *Connections* mit Applikationen,
      - *Message Manager*: für Remote *Communications*,
      - *Data Manager*: physische Speicherung der Nachrichten,
      - *Lock Manager*: Sperren von Queues und Nachrichten,
      - *Buffer Manager*: Pufferverwaltung und Abbildung auf Platte,
      - *Recovery Manager*: Fehlerbehandlung und Restart,
      - *Log Manager*: Recovery Log.

Queued Transaction Processing (9)

❑ Verwaltung von Multi-Transaction-Workflows

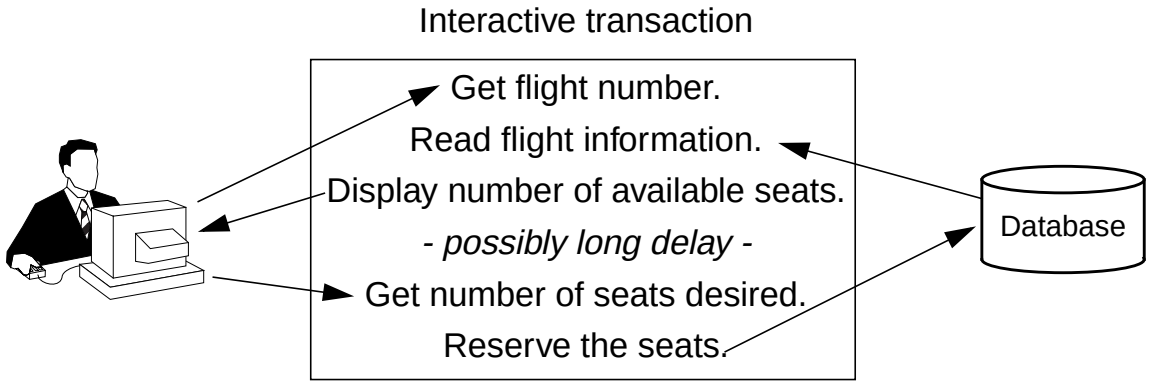


➪ Gründe für Multi-Transaction-Workflow

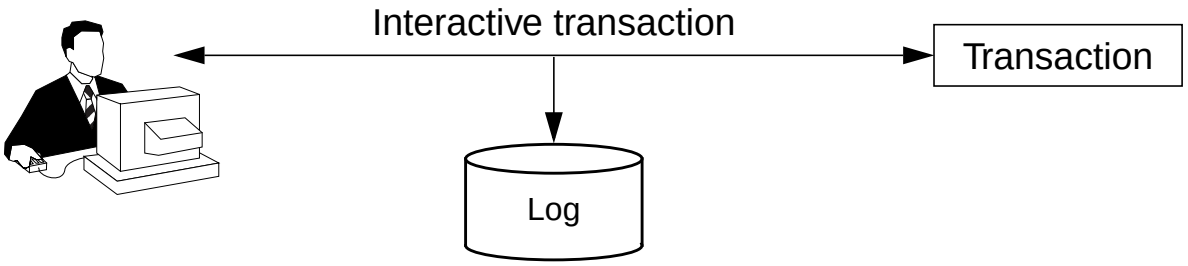
- Verfügbarkeit von Ressourcen
- Real-World-Constraints
- System-Constraints
- Kapselung von Funktionen
- Belegung von Ressourcen

Queued Transaction Processing (10)

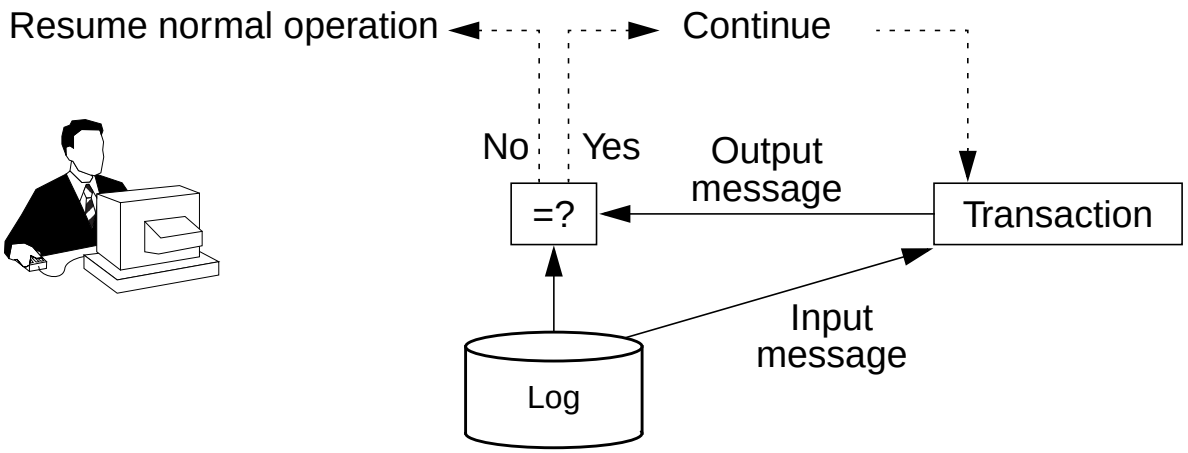
- Interaktive Mehr-Schritt-Aktivitäten ohne Queuing



  - *Pseudoconversations*
  - Logging von Eingabe-/Ausgabe-Operationen



a. During normal operation, log all messages.



b. Use the log to recover from a transaction failure.

Zusammenfassung (1)

- Kommunikation, insbesondere zwischen den Komponenten eines TP-Monitors, bei Abbildung auf verschiedene Prozesse:
  - (T)RPC
    - relativ einfach in der Benutzung
    - gewohnter Prozeduraufruf
    - transparente Behandlung fast aller Kommunikationsapekte
    - Nachteil: komplexer bzgl. System-Setup, Konfiguration, Leistung
  - Peer-to-Peer
    - hohe Flexibilität
    - fast beliebige Nachrichtenfolgen
    - LU6.2 als Kommunikationprotokoll für IBM's Peer-to-Peer-APIs als De-Facto-Standard zu betrachten (die meisten TP-Monitore bieten LU6.2-Gateway zur Kommunikation mit IBM-TP-Systemen)
    - Nachteil: hohe Komplexität in der Programmierung
  - Queuing
    - Entkopplung/Asynchronität unter Beibehaltung von Garantien
    - Lastbalancierung, Prioritäten
    - Queue-Manager muss Request-IDs verwalten, damit Client angemessene Recovery durchführen kann
    - *Multi Transaction Workflows*
    - bei interaktiven Transaktionen oft *Pseudoconversations* oder *Message Logging* ausreichend