Datenkommunikation

Studienarbeit

Wintersemester 2011/2012

Überblick

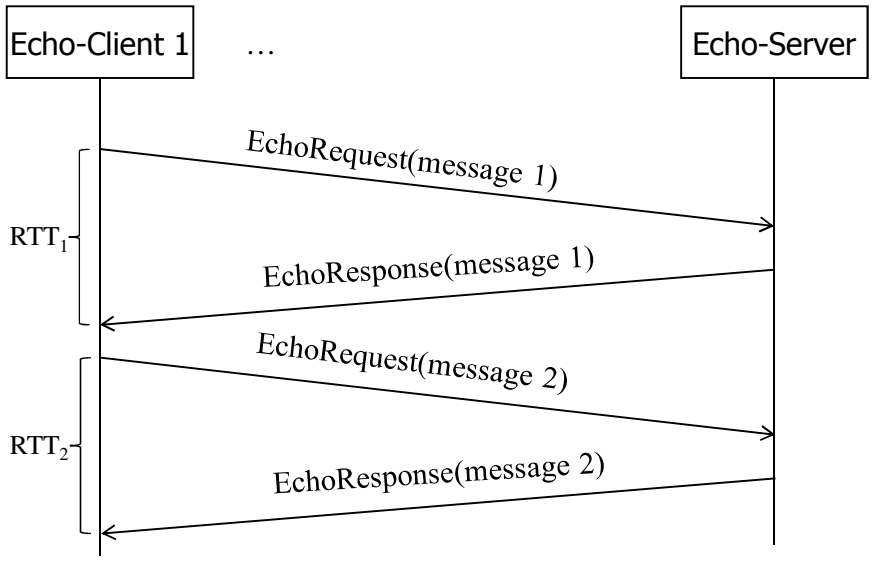
1. **Einführung in die Aufgabenstellung**
   - Überblick und Lernziele
   - Teilaufgaben 1 - 7
2. Details zur LWTRT-Schicht
   - Hinweise zur Protokollspezifikation
   - Hinweise zur Implementierung

Überblick über die Aufgabenstellung

- **Konzeption und Implementierung** einer Kommunikationssoftware
- Szenario in diesem Semester ist eine einfache **Echo-Anwendung** (Client-/Server)
- Nutzung *mehrerer Transportprotokolle* (TCP, UDP, LWTRT, RMI), Transportzugriff über **Sockets**
- Programmieren in einer höheren Programmiersprache → wir verwenden **Java**
- **Leistungsvergleich** für verschiedene Lösungen

Echo-Anwendung

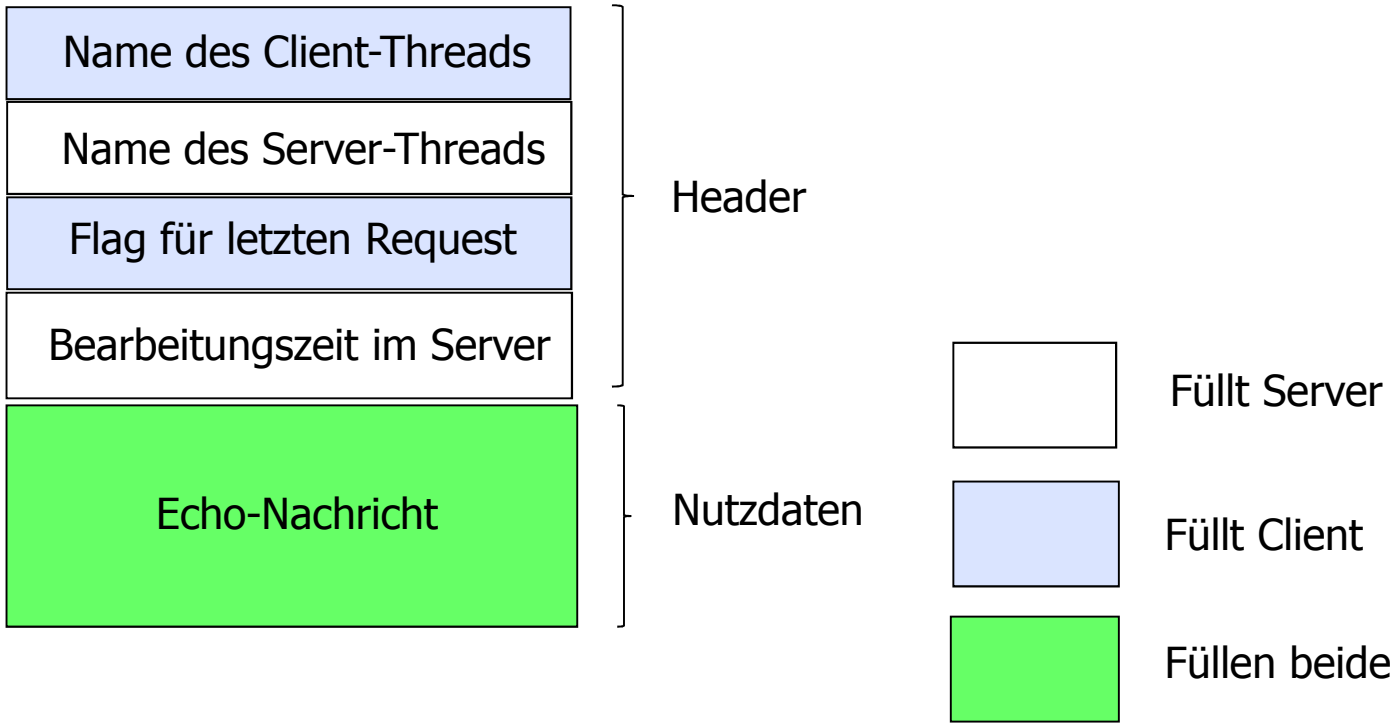
- Client sendet Anfrage (Echo-Request)
- Server registriert Client, falls noch nicht erfolgt, in einer „Client-Liste“
- Server antwortet mit Response (Echo-Response)
- Nachrichtenformat ist definiert → Echo-PDU



RTT = Round Trip Time

Echo-Protokoll (1)
Nachrichtenaufbau

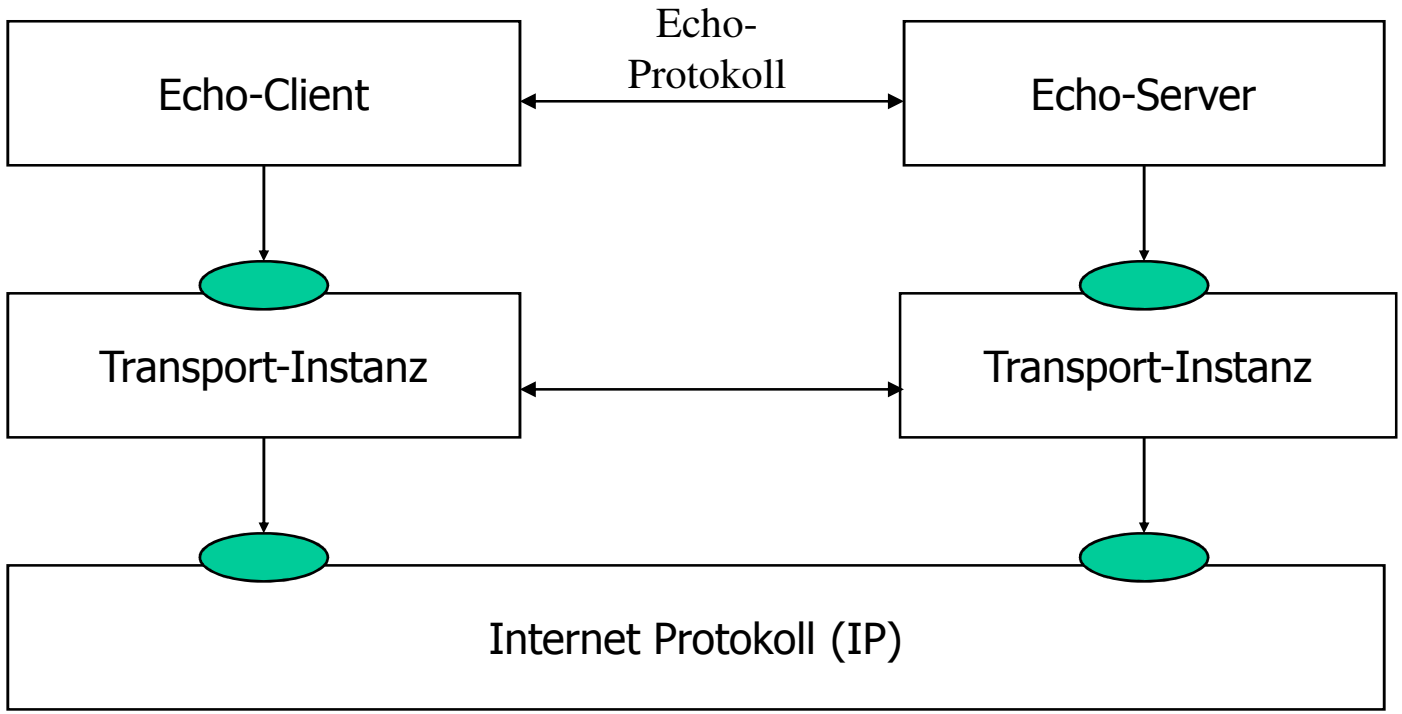
- Einfaches Nachrichtenformat in Klasse EchoPDU definiert:



Echo-Protokoll (2): Regelwerk

- Name des Client-Threads wird durch den Client belegt
  - Auf Eindeutigkeit achten, jeder Client-Thread erhält eigenen Namen → Thread.setName()
- Name des Server-Threads wird durch den Server belegt
  - Jeder Server-Thread erhält einen eigenen Namen
- Flag für letzten Request: Angabe des Client-Threads, ob es seine letzte Echo-PDU ist (Letzter Request = true)
  - Wird vom Server verwendet, damit der Client wieder aus der Client-Liste gelöscht werden kann
- Bearbeitungszeit im Server:
  - Zeit, die der Server für die Bearbeitung des Request benötigt (Zeitmessung in Nanosekunden)
- Echo-Nachricht: Eigentliche Nutzdaten

Grobe Schichtenarchitektur



Lernziele

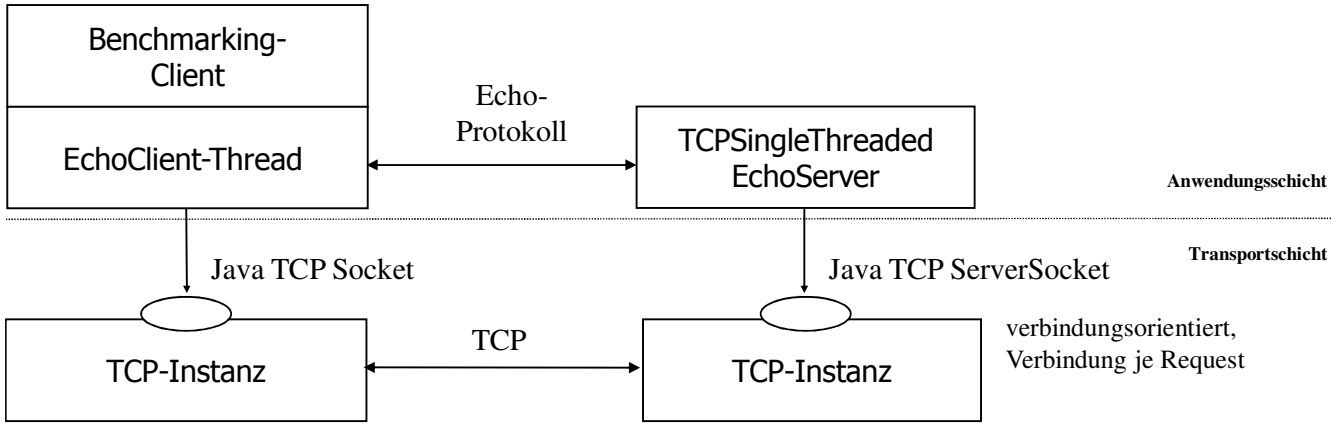
- **Erfahrungen** in der Programmierung von Kommunikationsanwendungen (hier: Client-/Server-Anwendungen) **sammeln**
- **Schichtenorientiertes Denken** schulen
- Problemstellungen der **Protokoll-implementierung** praktisch erfahren
  - Komplexität anderer Protokolle soll verständlicher werden
  - Methodisches Vorgehen beim Vergleich von Lösungsansätzen insbesondere in der Leistungsanalyse

Single-threaded versus Multi-threaded Server

- Serverseitig gibt es große Unterschiede in der Implementierung der **nebenläufigen** Verarbeitung
- **Single-threaded**: Im Wesentlichen übernimmt ein Thread die Bearbeitung aller ankommenden Requests von Clients
- **Multi-threaded**: Jeder Client oder jeder Request wird in einem eigenen Thread bearbeitet
- Weitere Unterscheidung: Verbindungsorientiert oder verbindungslos

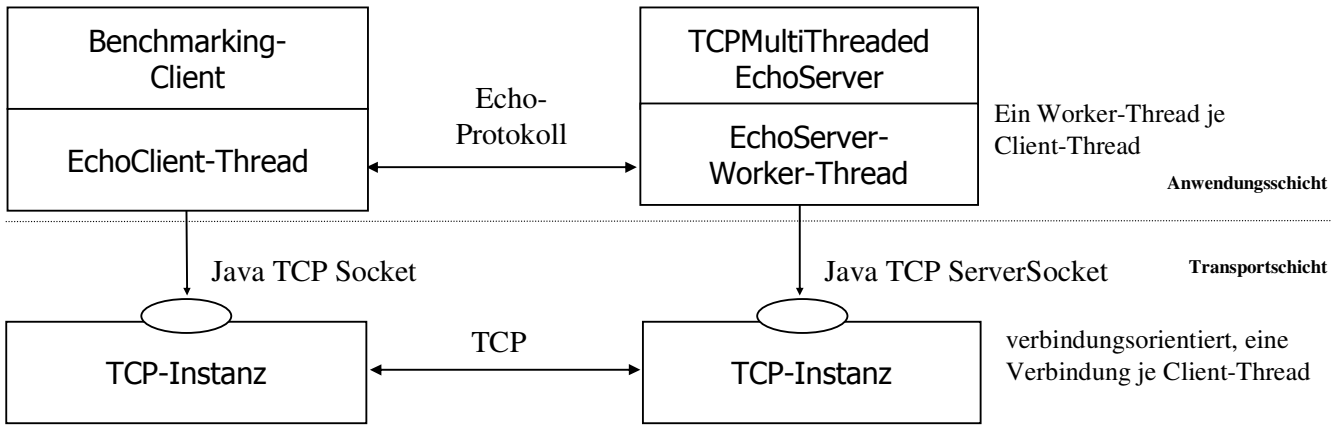
Teilaufgabe 1: Single-threaded TCP-Server

- Aufbau einer TCP-Verbindung zwischen Client und Server **für jeden Echo-Request** mit anschließendem (sofortigem) Abbau der Transportverbindung
- **Benchmarking-Client** zur Parametrisierung wird hier gleich mit entwickelt
  - **Datensammlung** für Leistungsvergleich einbauen (Basisklassen vorhanden)



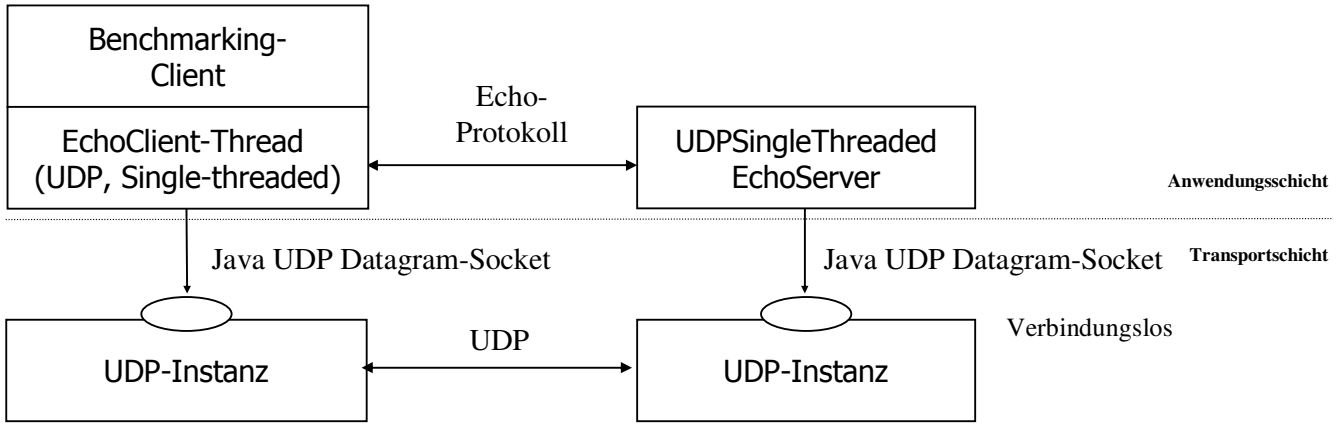
Teilaufgabe 2: Multi-threaded TCP-Server

- Serverseitig eigener „**Worker-Thread**“ für jeden Client-Thread
- Transportverbindung zwischen Client und Server wird am Anfang aufgebaut
- Transportverbindung bleibt bis zum Beenden eines Client-Threads bestehen



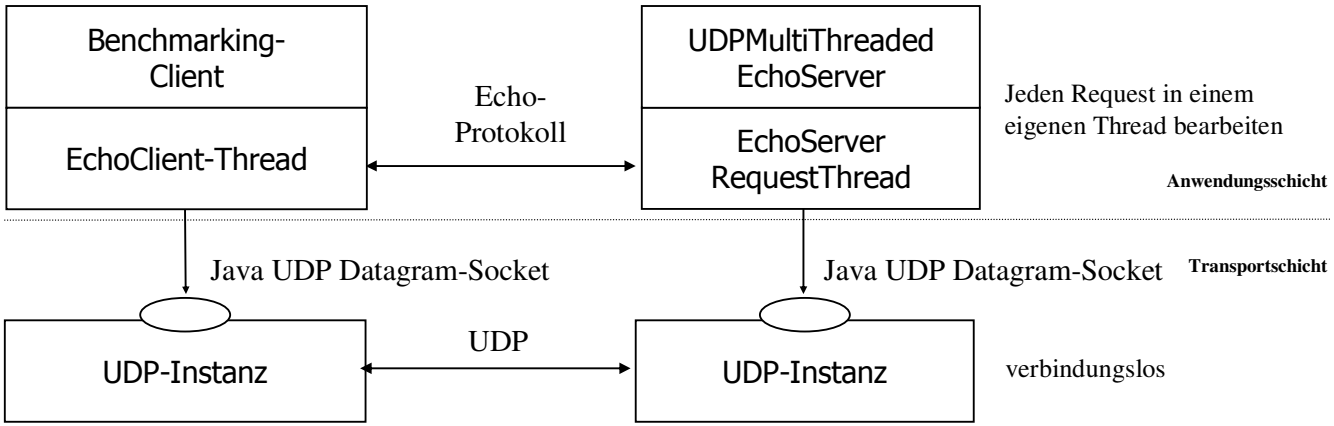
Teilaufgabe 3: Single-threaded UDP-Server

- **Ein Thread im Server**, der alle Echo-Requests bearbeitet
- **Verbindungslose** Kommunikation auf Basis von UDP-Datagram-Sockets



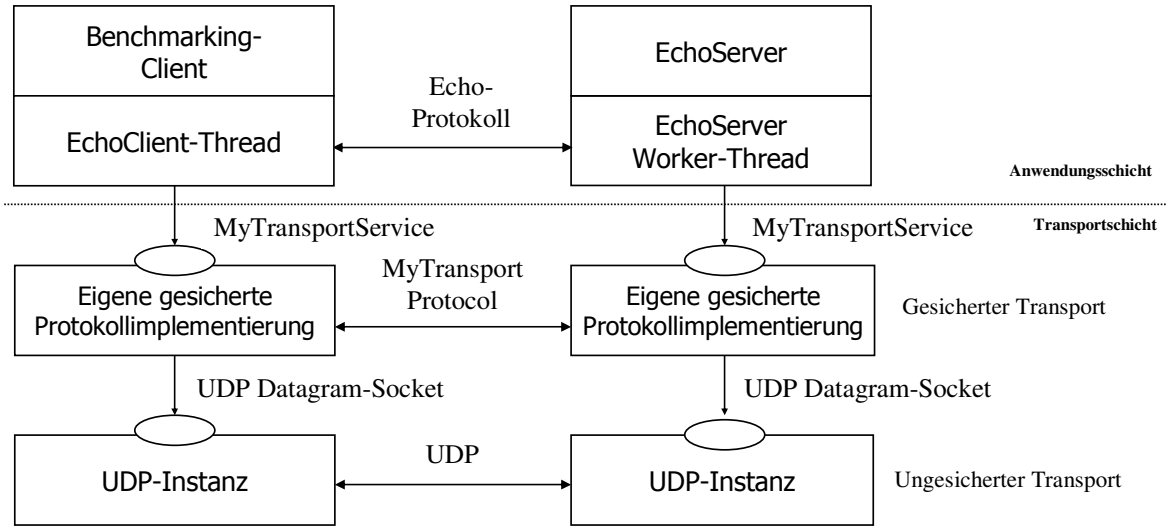
Teilaufgabe 4: Multi-threaded UDP-Server

- **Jeder Echo-Request** wird im Server in einem eigenen „**Worker-Thread**" bearbeitet
- **Verbindungslose** Kommunikation auf Basis von UDP-Datagram-Sockets



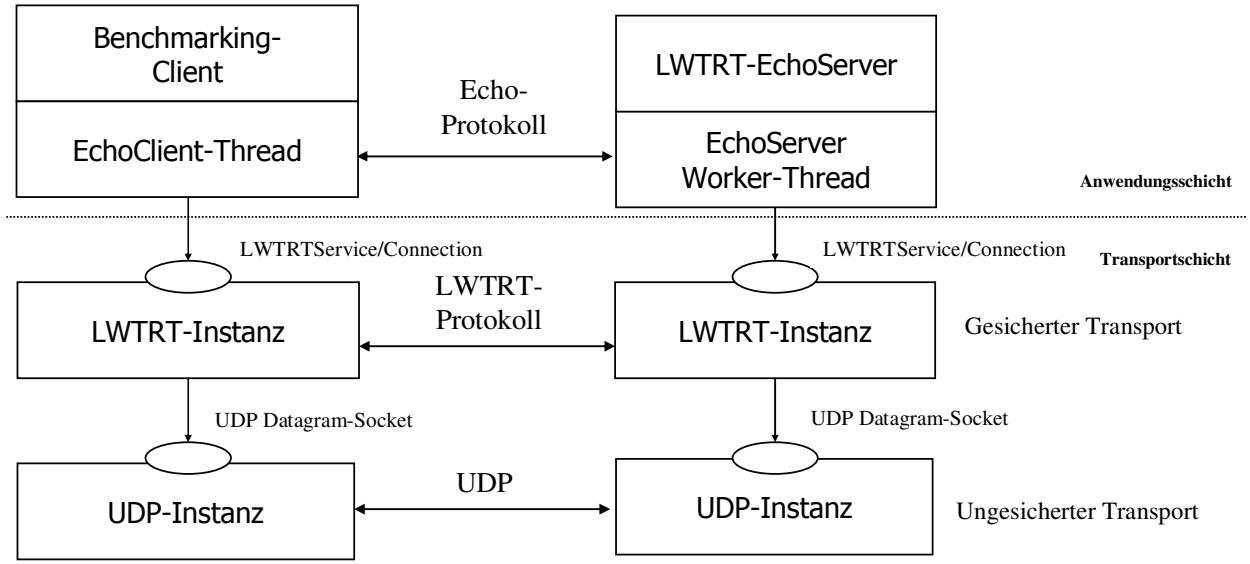
Teilaufgabe 5a: Eigene Implementierung (alternativ zu 5b)

- **Eigenentwicklung** eines gesicherten Transportprotokolls auf Basis von UDP
  - Einfaches Stop-and-Wait-Protokoll mit positiv-selektiver Quittierung
  - Timerüberwachung für alle Nachrichten
  - Ausfall eines Partners wird erkannt



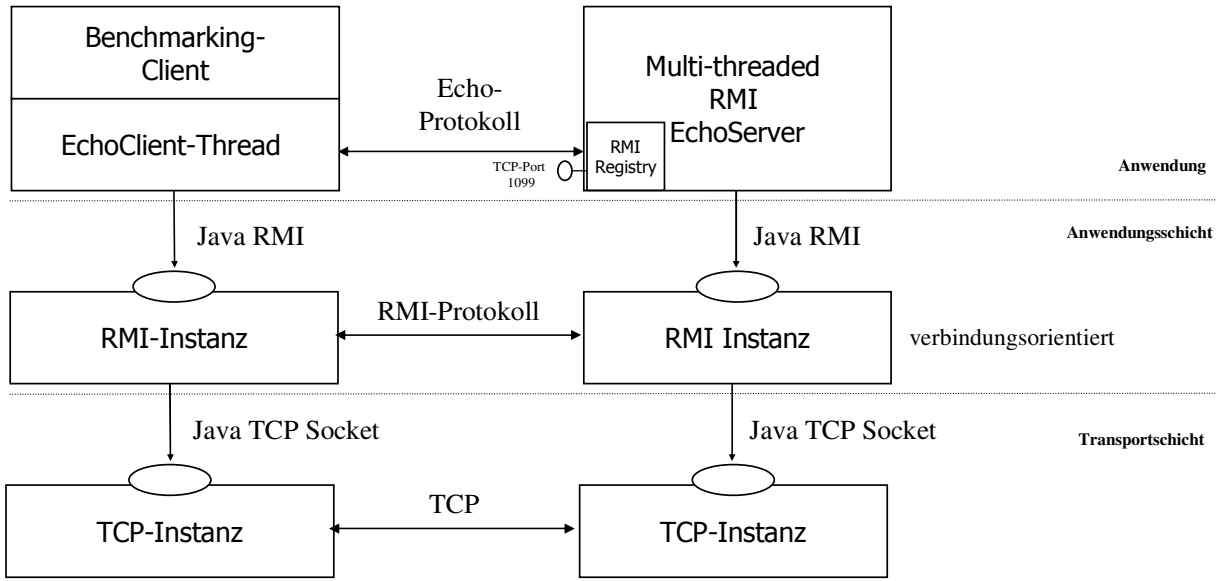
Teilaufgabe 5b: Multi-threaded LWTRT-Server (alternativ zu 5a)

- **Ergänzen** der vorhandenen LWTRT-Implementierung (siehe weiter unten) → Leichtgewichtiges Transportprotokoll (nicht perfekt)
- Vorhandene **LWTRT-Schnittstellen** nutzen und nicht verändern



Teilaufgabe 6: Multi-threaded RMI-Server

- Java RMI = Remote Method Invocation ist ein **höherwertiger** Client-/Server-Mechanismus und setzt auf TCP auf

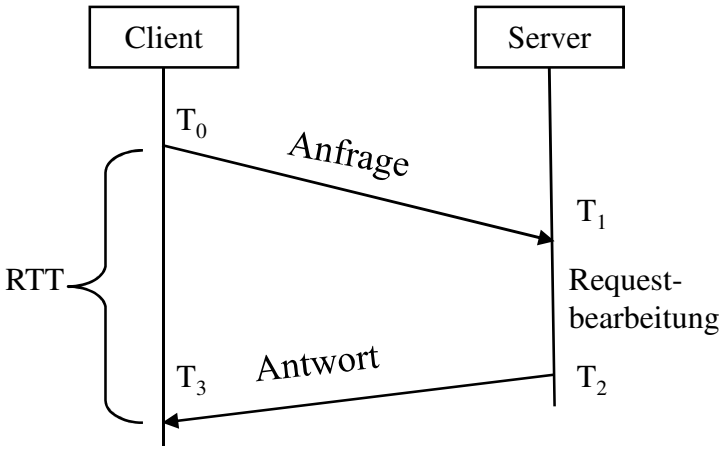


Teilaufgabe 7: Leistungsbewertung und Vergleich

- Verwendete Metrik: **RTT**
  - Serverbearbeitungszeit soll auch ermittelt werden
- Testdurchführung mit allen Implementierungen **mehrmals** (mind. 5 mal) **wiederholen**
- RTT-Berechnung: Mittelwert, Minimum, Maximum (über vorgegebene Klasse *SharedClientStatistics*)
- Leistungsauswertung in zwei Sichten durchführen und dokumentieren

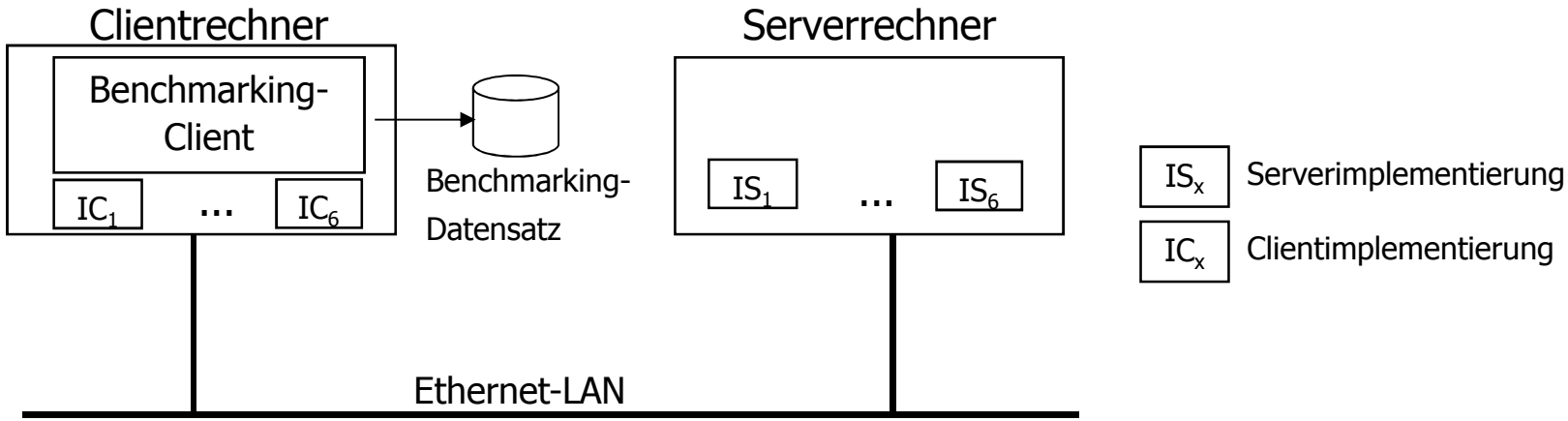
Teilaufgabe 7:
Metriken: RTT

- Definition:
  - Unter der Round-Trip-Time versteht man die Reaktions- oder Bearbeitungszeit eines Anwendungssystems für eine Anfrage
  - RTT bezeichnet die Zeitspanne, die erforderlich ist, um eine Nachricht bzw. einen Request von einem Sender zu einem Empfänger zu senden und die Antwort (den Response) des Empfängers wieder im Sender zu empfangen.
  - Messung in Millisekunden
  - $RTT = T_3 - T_0$



Teilaufgabe 7:
Aufbau der Messumgebung

- 1 Clientrechner zur Simulation der Clients (Client-Threads) mit Benchmarking-Client
- 1 Serverrechner
- Unbelastetes Netzwerk und unbelastete Rechner verwenden
- Testumgebung genau beschreiben: Rechnerausstattung, Netzwerk,... → Nachvollziehbarkeit des Benchmarks!!



Teilaufgabe 7:
Sammlung der Messdaten

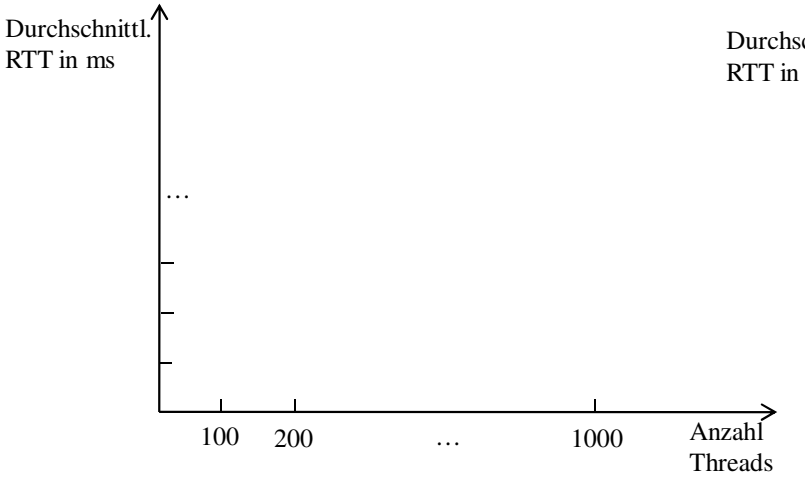
- RTT-Daten in Client-Threads bei jedem Request sammeln
- Abspeichern der Messergebnisse in einer Datei

  → vordefinierte Java-Klasse Nutzen
- Mehrfaches Wiederholen jedes Benchmarks (Achtung: zeitintensiv!)
- Daten anschließend auswerten

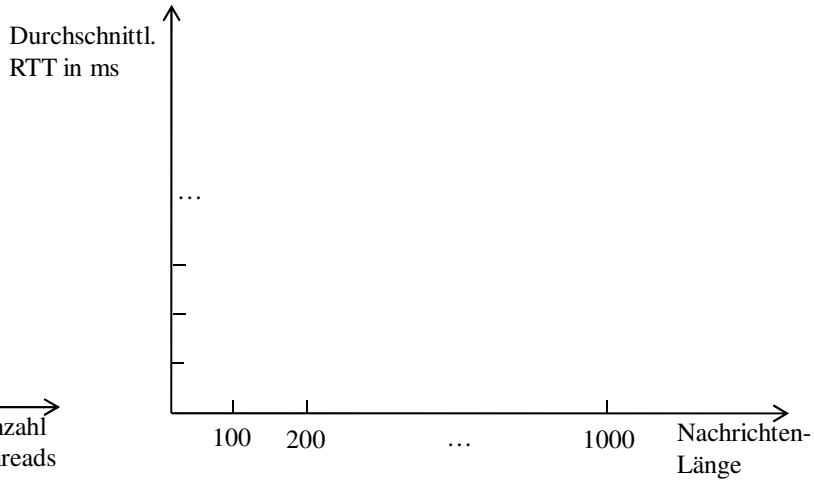
Teilaufgabe 7:
Messungen: Auswertung

- Zwei Grafische Darstellungen (über Excel)
  - Veränderung der Anzahl Client-Threads
  - Veränderung der Nachrichtenlänge

Messung Typ 1: Variation der Threads
- Nachrichtenlänge: 50 Byte
- Anzahl Nachrichten je Client: 100
- Denkzeit: 100 ms



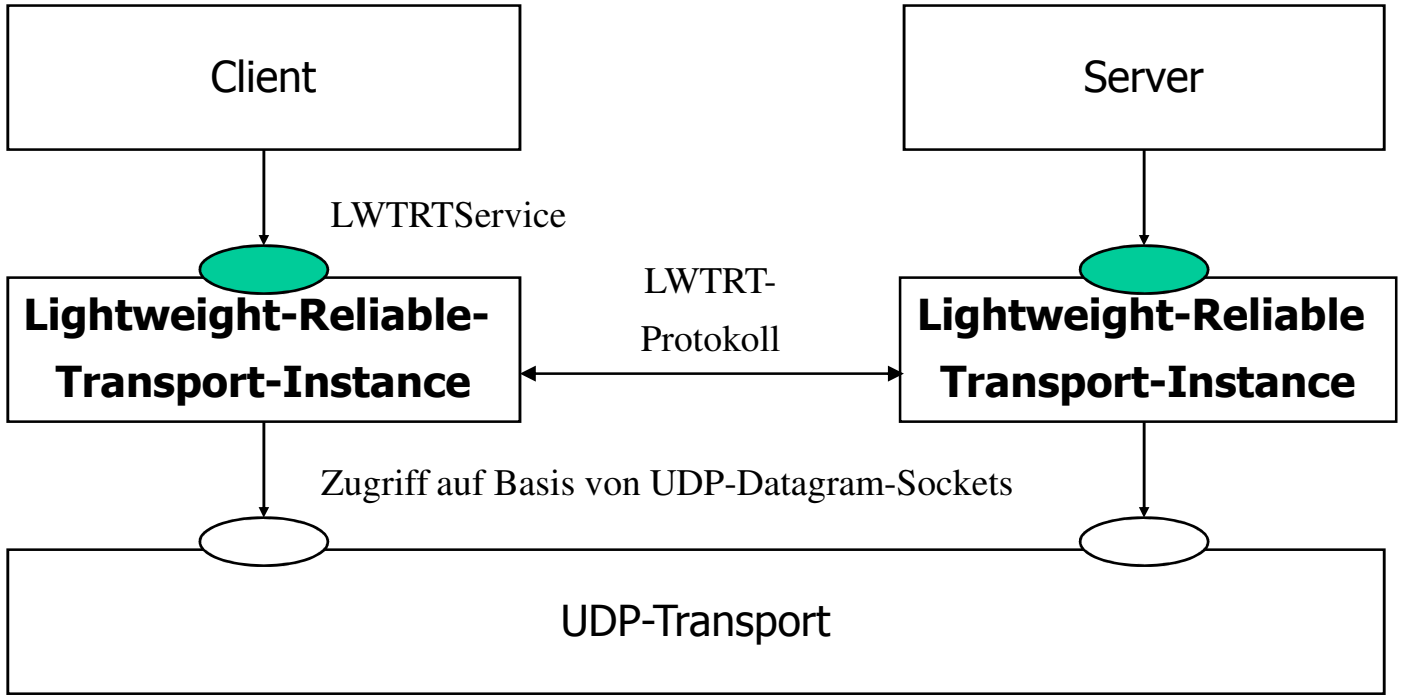
Messung Typ 2: Variation der Nachrichtenlänge
- Anzahl Nachrichten je Client: 100
- Anzahl Clients: 500
- Denkzeit: 100 ms



Überblick

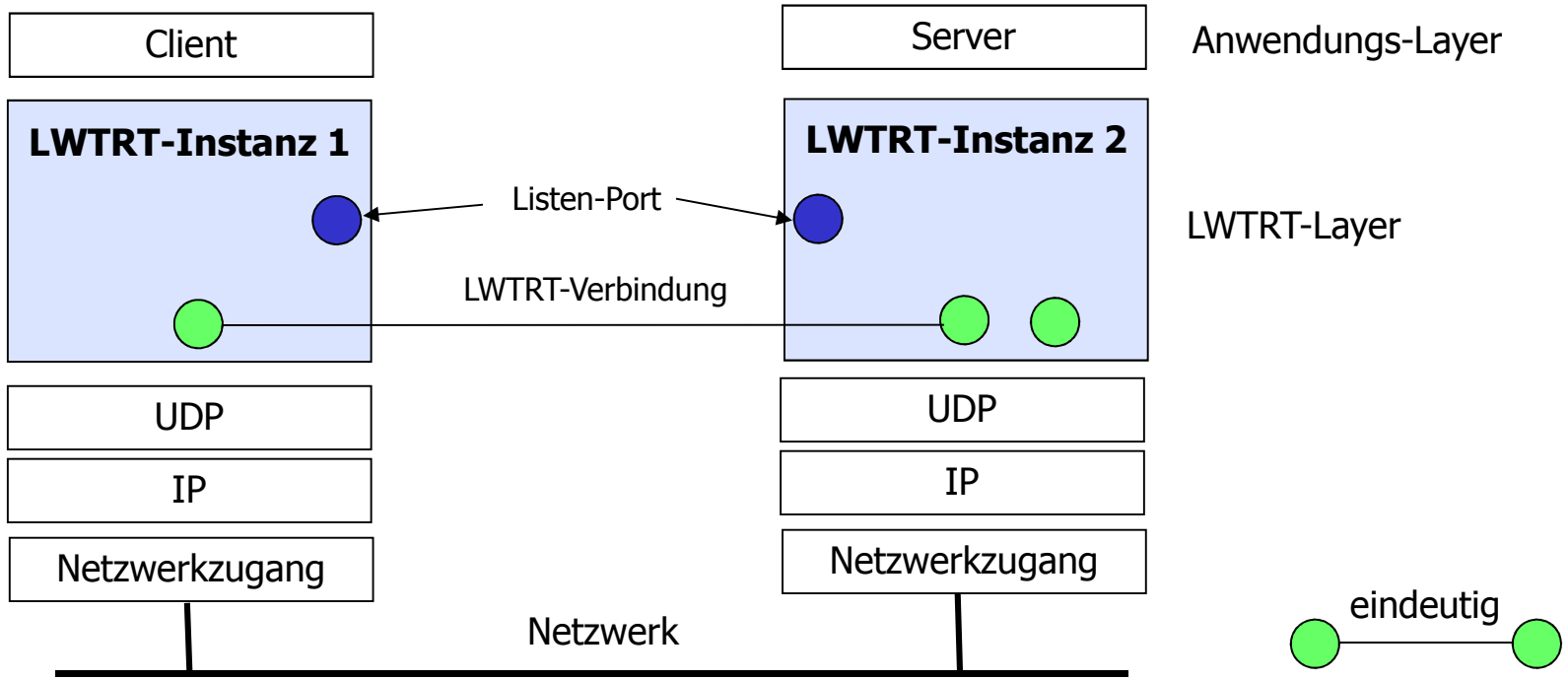
1. Einführung in die Aufgabenstellung
   - Überblick und Lernziele
   - Teilaufgaben 1 - 7
2. **Details zur LWTRT-Schicht**
   - Hinweise zur Protokollspezifikation
   - Hinweise zur Implementierung

Lightweight-Reliable-Transport-Protokoll



Protokollstack und Verbindungskonzept

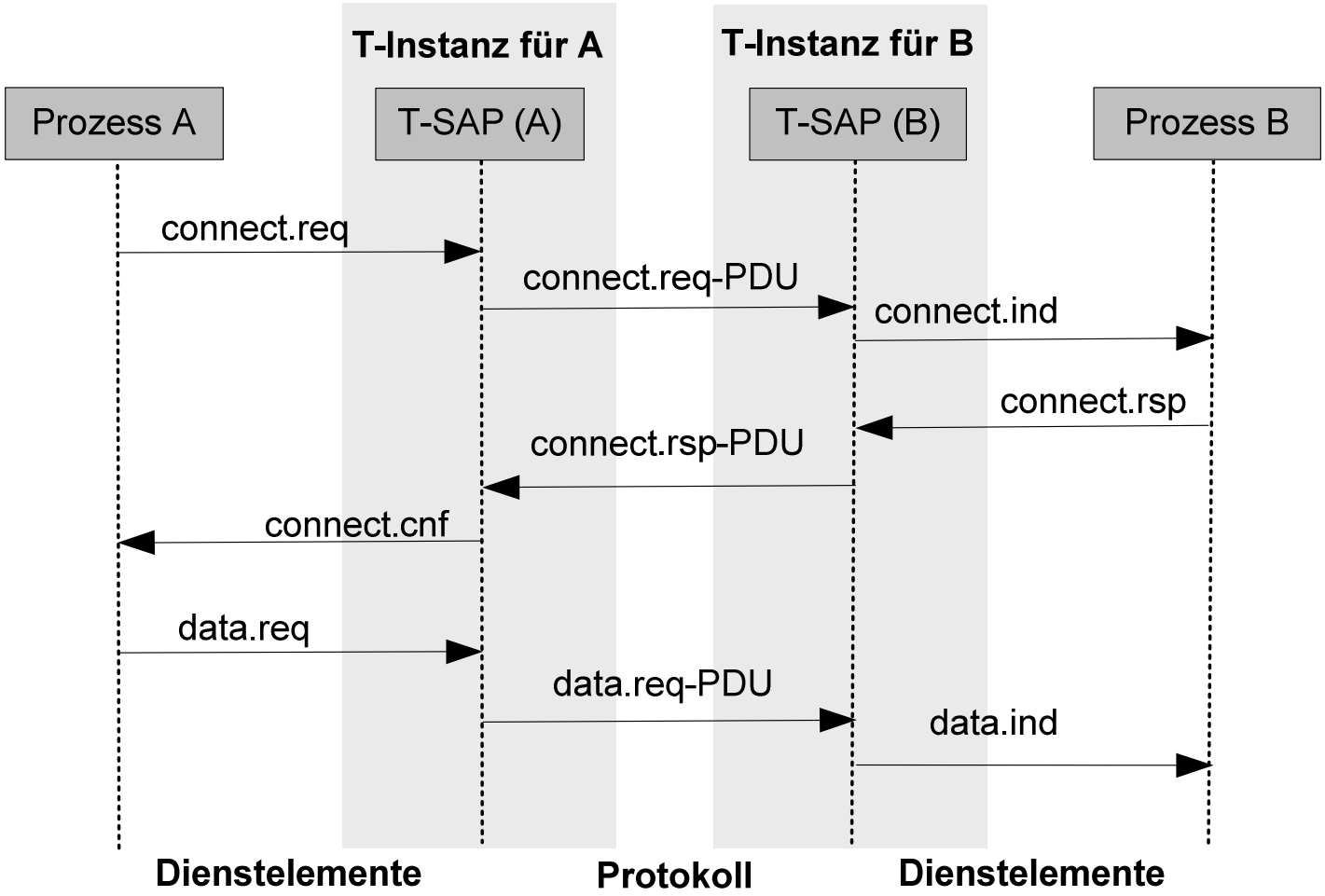
- LWTRT stellt eine zuverlässige Ende-zu-Ende-Verbindung bereit
- Jede LWTRT-Instanz horcht an einem UDP-Port auf Verbindungsaufbauanfragen
- Eine LWTRT-Instanz kann beliebig viele Verbindungen unterhalten



Exkurs:
Dienste und Dienstprimitive (= Dienstelemente)

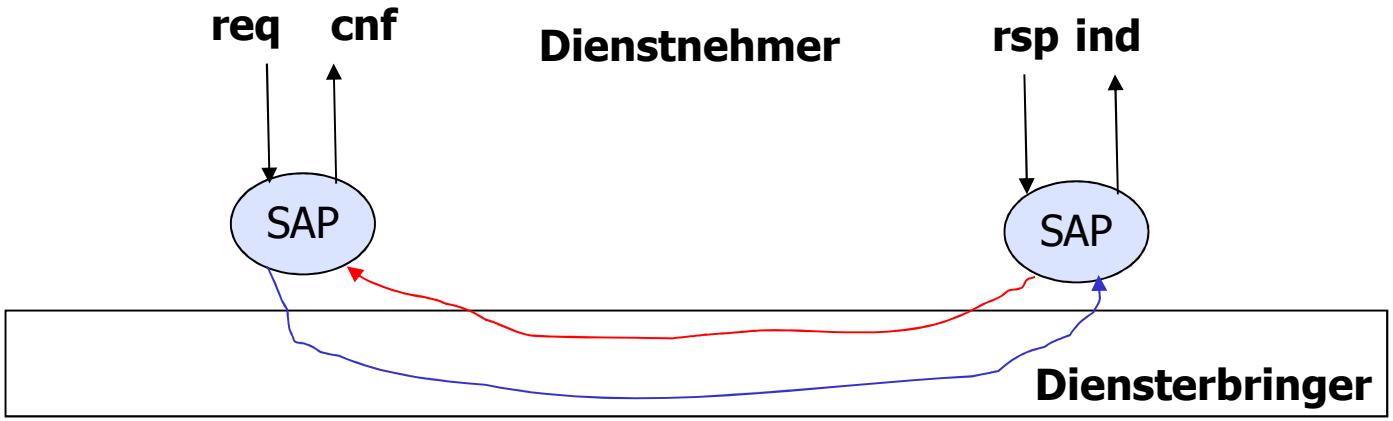
- Dienste (Abstraktion) im ISO/OSI-Modell
  - Beispiel: connect, disconnect, data (Datenübertragung)
- Dienstelemente/Dienstprimitive sind Operationen eines Dienstes
- Typische Dienstprimitive sind
  - Request
  - Indication
  - Confirmation
  - Response
  - Beispiel: connest.req, connect.ind, data.req

Exkurs:
Dienste und Protokoll (Zusammenspiel)



Exkurs: Dienstelemente/Dienstprimitive

- Dienstelemente eines Protokolldienstes:
  - req = Request, Anforderung vom Dienstnehmer
  - rsp = Response, Antwort vom Dienstnehmer
  - ind = Indication, Anzeige vom Diensterbringer
  - cnf = Confirmation, Bestätigung vom Diensterbringer



LWTRT-Protokoll und -Dienst

- „Zuverlässiger" Transportdienst
- Zuverlässig im Vergleich zu UDP
- Bei weitem nicht so komplex wie TCP
- Einfache Protokollmechanismen zur Erhöhung der Zuverlässigkeit:
  - Bidirektional und vollduplexfähig
  - Einfaches Stop-and-Wait-Protokoll mit positiv-selektiver Quittierung
  - Timerüberwachung für alle Nachrichten
  - Ausfall eines Partners wird erkannt
  - PDUs werden serialisiert
    - Vereinfachung: Als Java-Objekte

LWTRT-Protokoll und -Dienst: Dienstübersicht (Dienste und Dienstprinitive)

- Register (req, cnf)
  - Registrieren einer Anwendung und Listenport aktivieren
- Unregister (req, cnf)
  - Listenport freigeben und alle Verbindungen auflösen
- Connect (req, ind, rsp, cnf)
  - Verbindung aufbauen
- Disconnect (req, ind, rsp, cnf)
  - Verbindung auflösen
- Data (req, ind, rsp, cnf)
  - Datenübertragung
- Ping (req, ind, rsp, cnf) (optional)
  - Prüfen, ob Verbindung aktiv ist

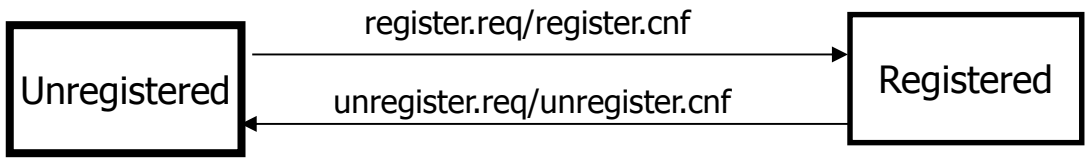
Achtung:

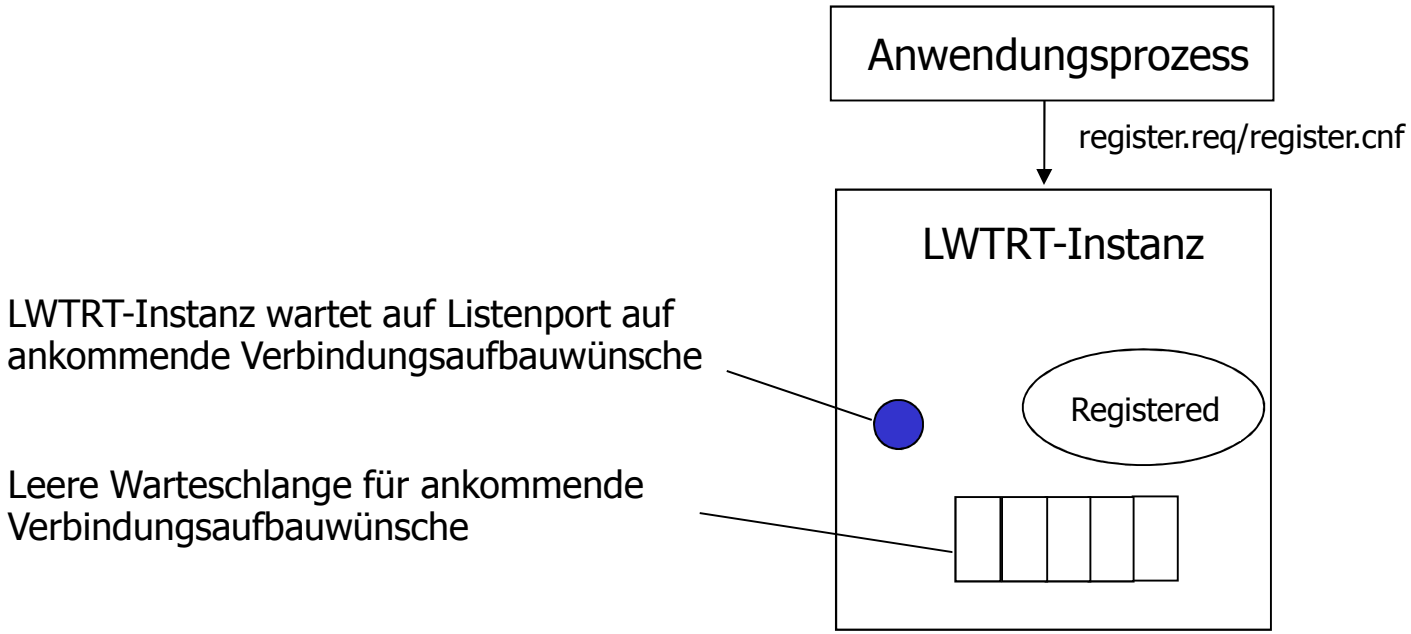
- Das sind keine Java-Methodennamen

LWTRT-Spezifikation: Registrierung

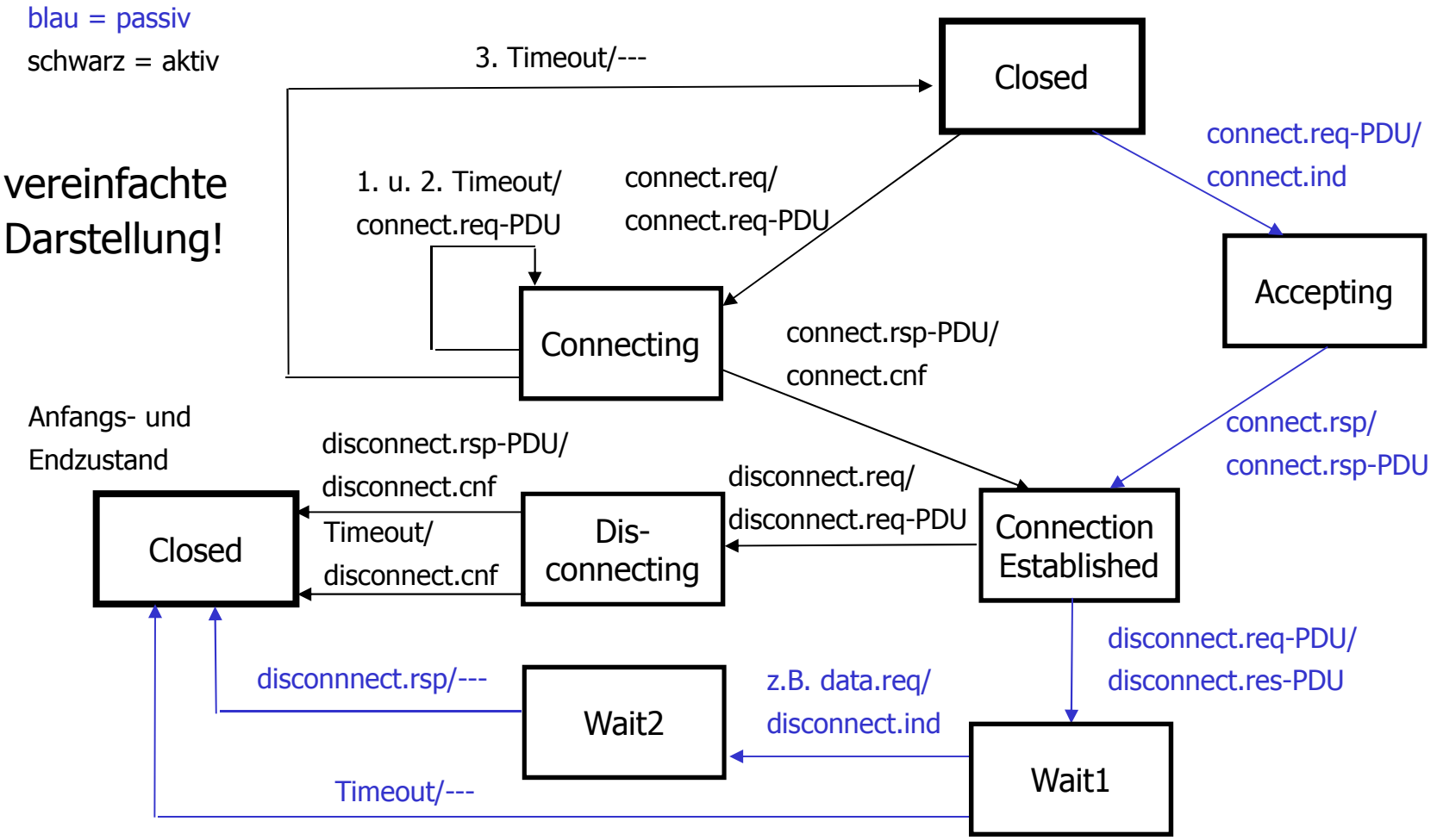
- Register-Dienst: lokaler Dienst
  - LWTRT-Schicht wird eingerichtet
  - Anwendung wird zum Entgegennehmen von Verbindungsaufbauwünschen vorbereitet
  - Verbindungswarteschlange wird eingerichtet
- Unregister-Dienst: lokaler Dienst
  - Verbindungswarteschlange wird abgebaut und es wird kein Verbindungswunsch mehr akzeptiert
  - Alle Verbindungen werden lokal aufgelöst
  - Partner werden nicht informiert (vereinfacht) und merken Verbindungsauflösung erst beim nächsten Dienstaufruf für die jeweiligen Verbindungen

LWTRT-Zustandsautomat: Registrierung





LWTRT-Zustandsautomat: Verbindungsmanagement

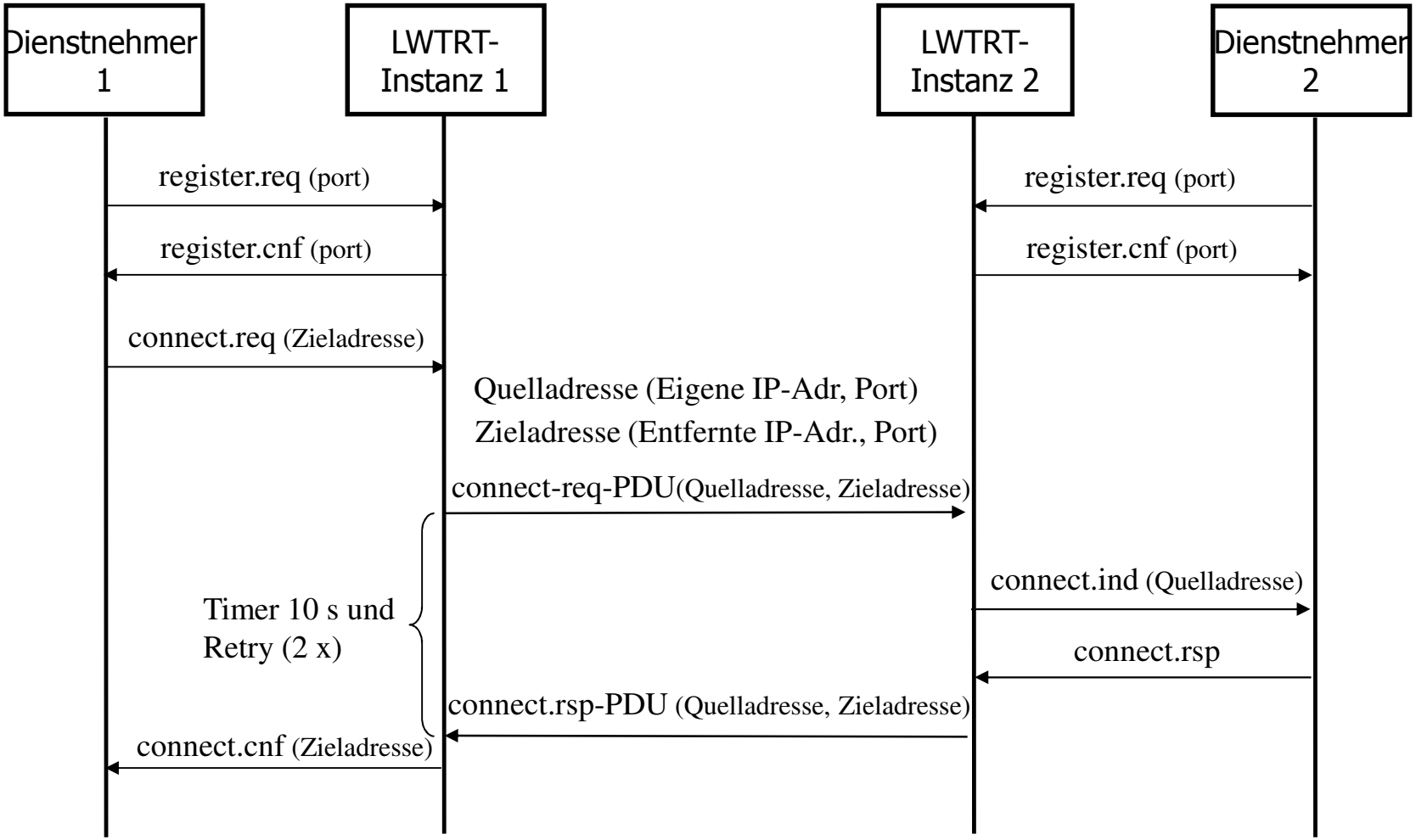


Je Verbindung ein Zustandsautomat bei jedem Partner

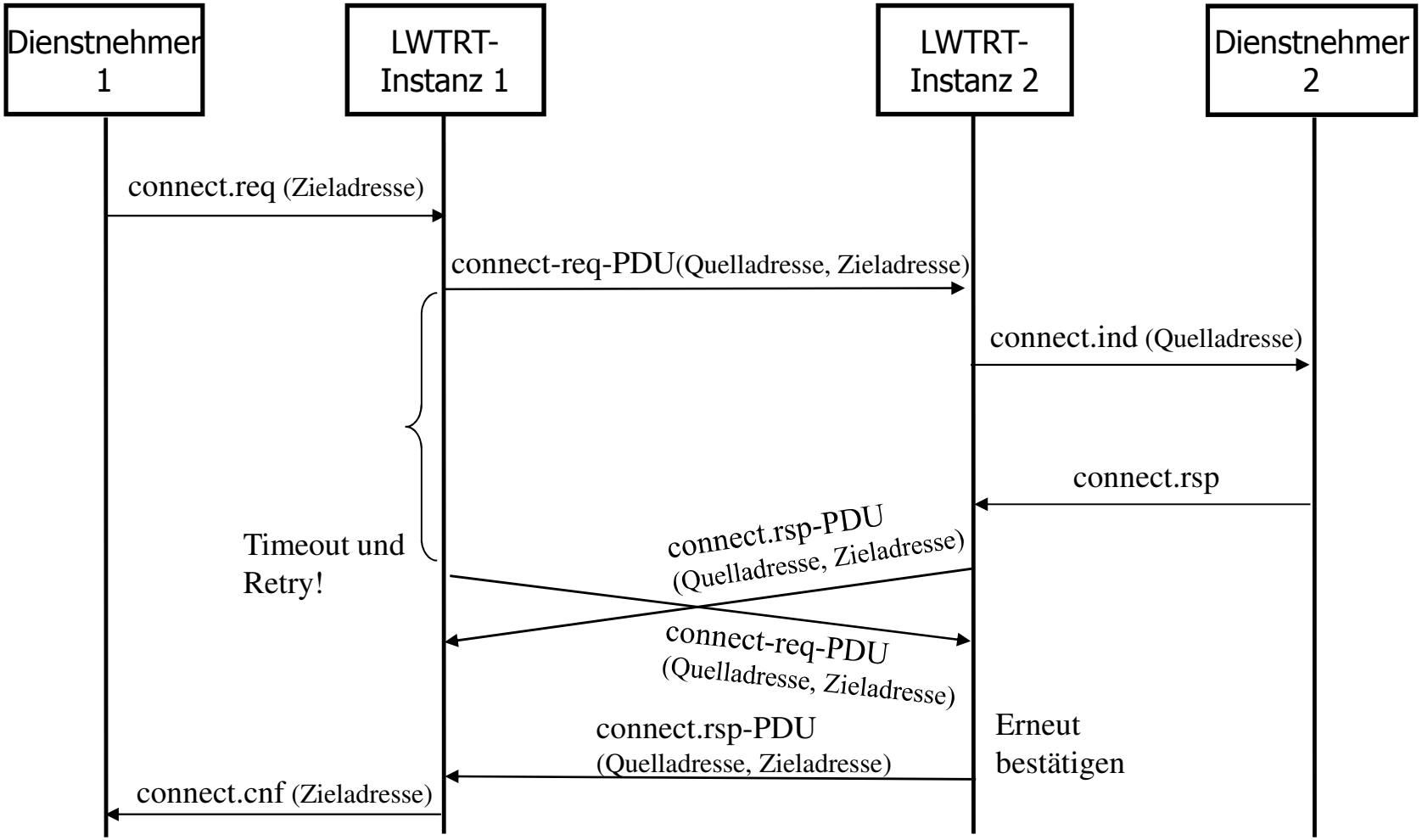
LWTRT-Spezifikation: Verbindungsaufbau

- Connect-Dienst:
  - Verbindungsaufbauwunsch wird vom aktiven Partner gesendet, maximal 2 x wiederholen bei Timerablauf (10 s)
  - Jede Verbindung ist durch ein eindeutiges Socket-Pair global identifiziert (Zieladresse = IP-Adr + UDP-Port, Quelladresse = dito)
  - Verbindungsaufbauwunsch wird vom passiven Partner aus Verbindungswarteschlange entgegengenommen
  - Verbindungsaufbauwunsch kann vom passiven Partner nicht explizit abgelehnt werden (Vereinfachung)
  - Wenn die Verbindung schon steht und es kommt erneut ein Verbindungsaufbauwunsch für diese Verbindung an, wird nochmals eine connect.rsp-PDU gesendet
  - **Spezialszenarien** sind zu überlegen:
    - Verbindung schon bestätigt und es kommt erneut ein Verbindungsaufbau-wunsch für diese Verbindung an

LWTRT-Spezifikation: Verbindungsaufbau



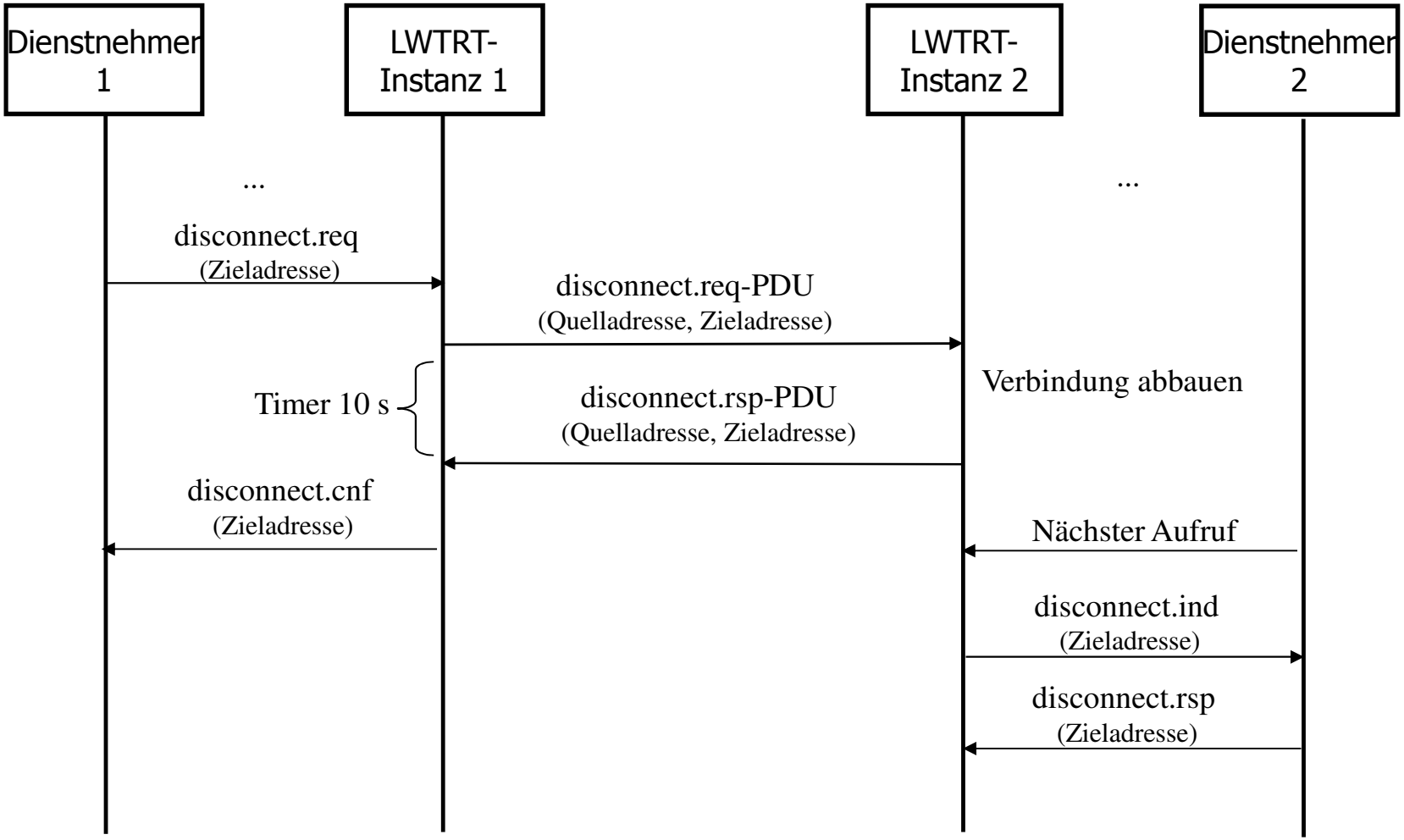
LWTRT-Spezifikation: Verbindungsaufbau - Spezialfall



LWTRT-Spezifikation: Protokollregeln zum Verbindungsabbau

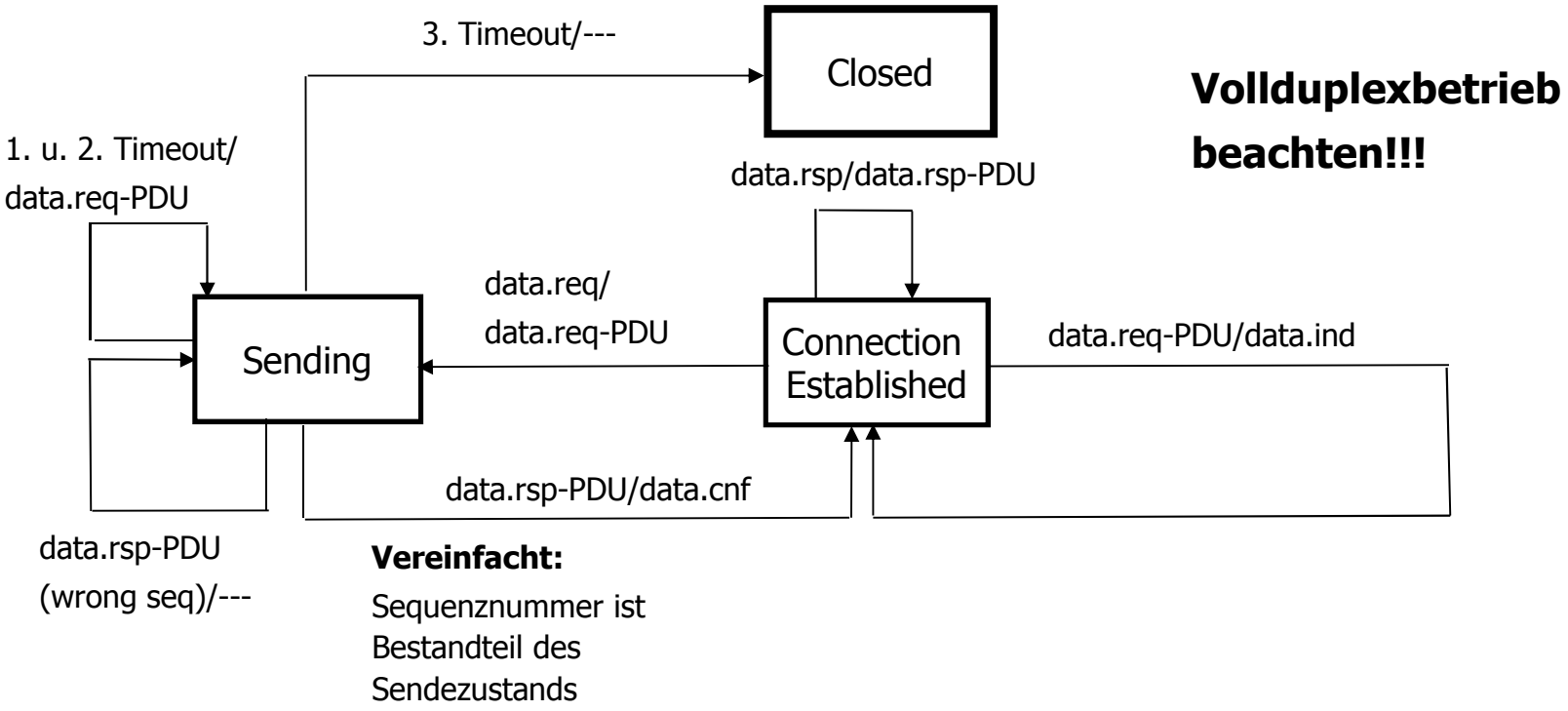
- Disconnect-Dienst:
  - Verbindungsabbau kann von beiden Seiten initiiert werden, auch von beiden quasi gleichzeitig
  - Die LWTRT-Instanz bestätigt den Verbindungsabbauwunsch und merkt sich lokal den Status
  - Erst beim nächsten Dienstaufruf des (für diesen Vorgang) passiven Partners wird Verbindungsabbau bemerkt
  - Spätestens nach 300 s wird das Socket-Pair auf der passiven Seite in jedem Fall freigegeben
  - Socket-Pair muss solange blockiert werden, bis beide Seiten den Verbindungsabbau abgeschlossen haben
  - Timerüberwachung auf der aktiven Seite: 10 Sekunden auf Bestätigung warten und danach die Verbindung auch ohne Bestätigung abbauen
  - Bis zum beidseitigen Verbindungsabbau, also bis zum Empfang der disconnect.rsp-PDU oder bis zum Timeout, müssen beim aktiven Partner noch alle bereits gesendeten Nachrichten im Zustand *Disconnecting* angenommen werden

LWTRT-Spezifikation: Verbindungsabbau



LWTRT-Zustandsautomat: Senden und Empfangen

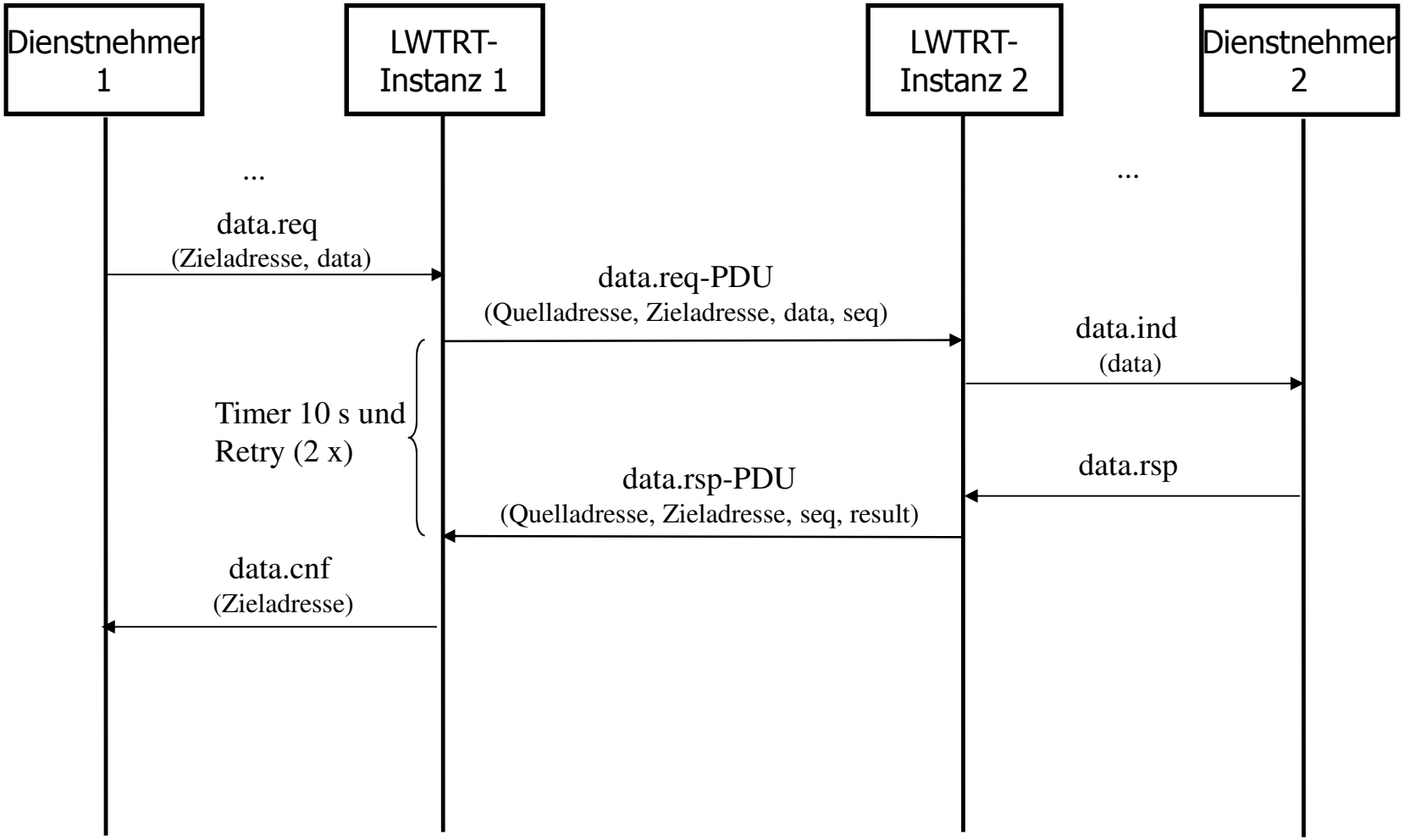
- Vereinfachte Darstellung



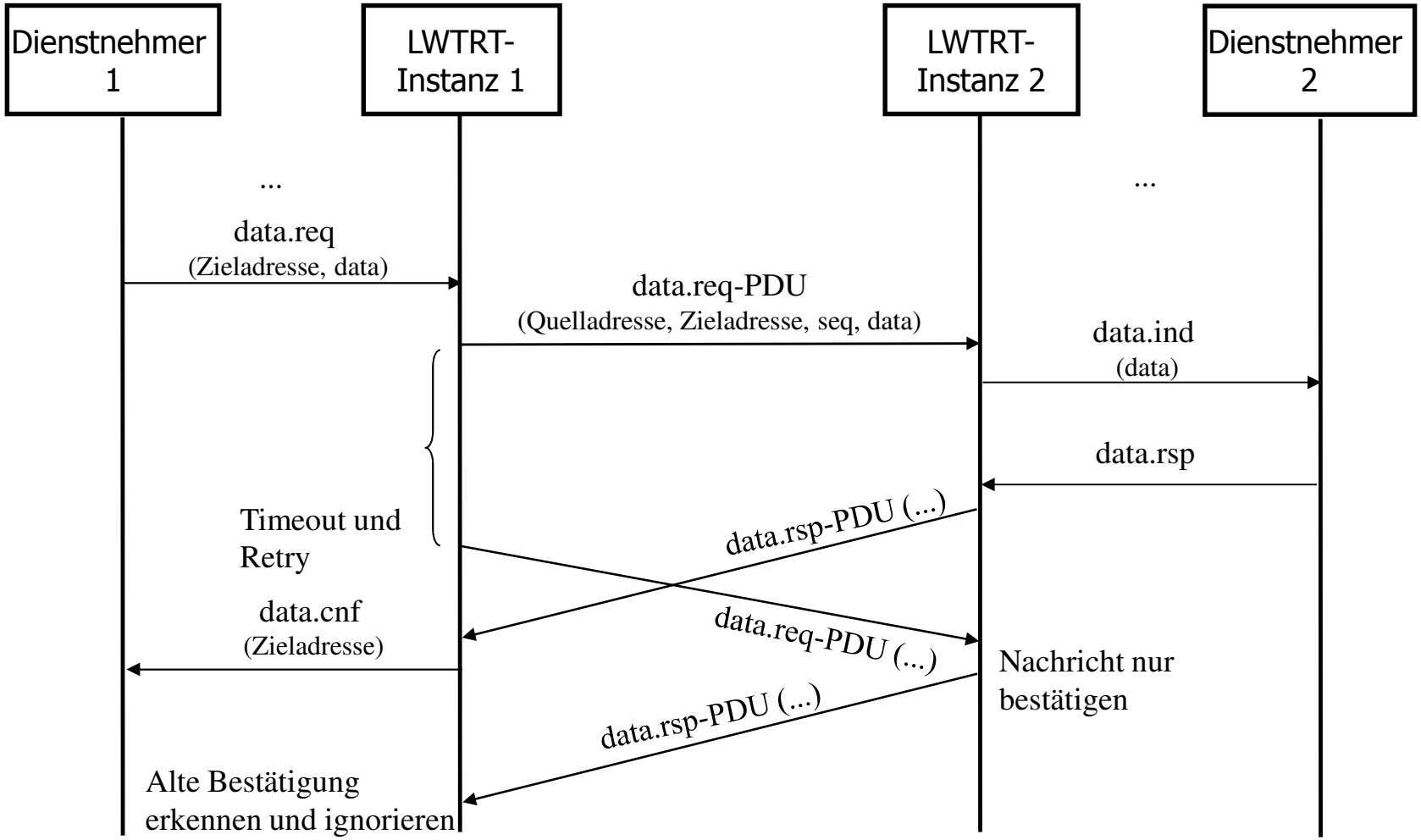
LWTRT-Spezifikation: Protokollregeln zur Datenübertragung

- Data-Dienst:
  - Es kann nur über eine bestehende Verbindung gesendet und empfangen werden
  - Jede Seite verwaltet eine Sequenznummer, Beginn bei 0
  - Timerüberwachung: Wenn data.rsp-PDU (ACK) nicht ankommt, wird nach 10 s wiederholt
  - Maximal 2 mal wiederholen, bei Misserfolg Abbau der Verbindung
  - Jede Nachricht wird nur maximal einmal zugestellt, Sendungswiederholungen werden an Sequenznummer erkannt
  - Kommt dieselbe Nachricht erneut an, wird sie ohne weitere Aktivitäten verworfen (Erkennung an Sequenznummer)
  - Die Bestätigung einer data.req-PDU wird über die Sequenznummer in der data.rsp-PDU erkannt
  - **Spezialszenarien** sind zu überlegen:
    - Nachricht wird wegen Timerablauf ein zweites Mal gesendet, wurde aber gerade vorher vom Partner bestätigt, Bestätigung ist noch nicht angekommen
    - ...

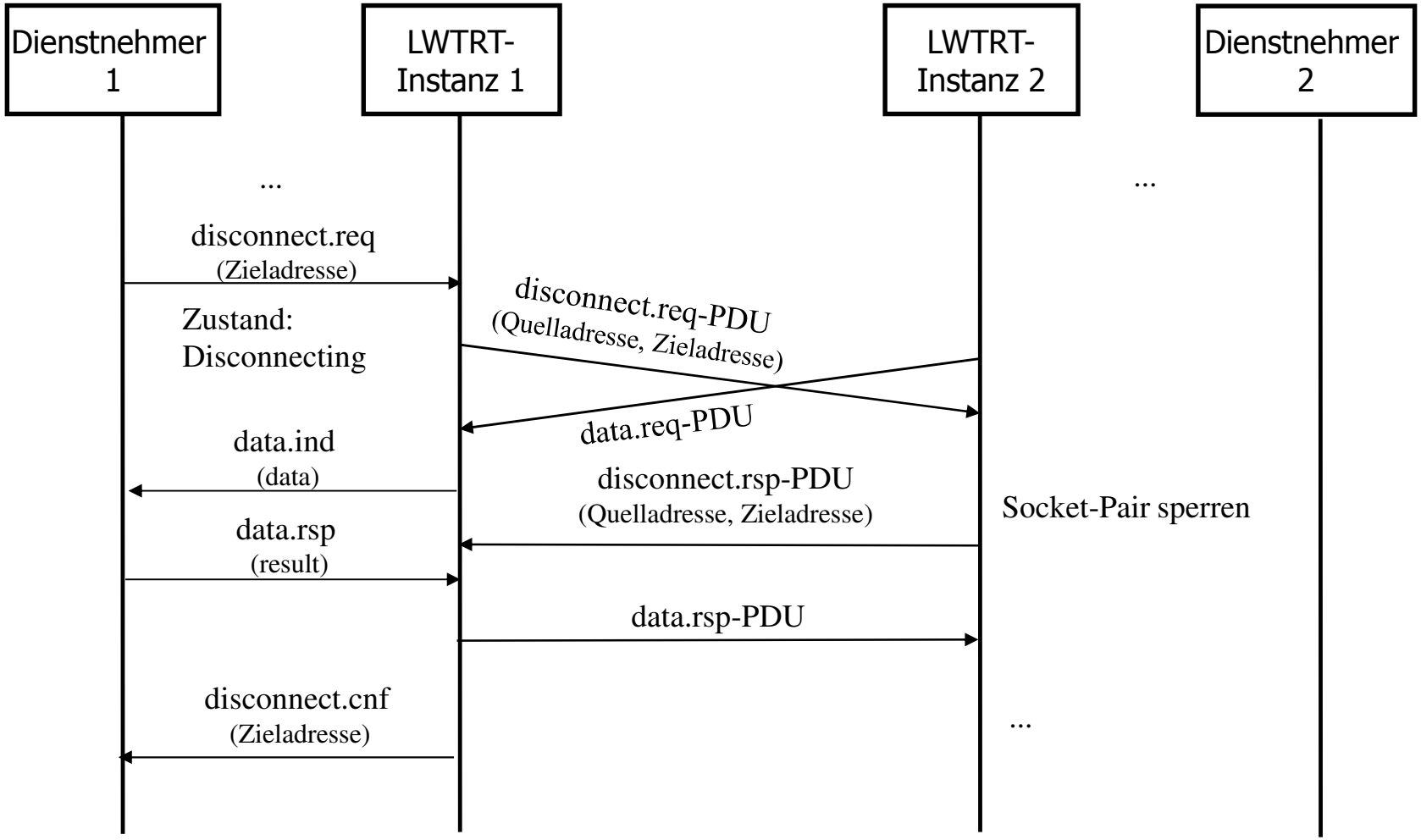
LWTRT-Spezifikation: Datenübertragung



LWTRT-Spezifikation:
Spezialfall: Timeout und erneutes Senden



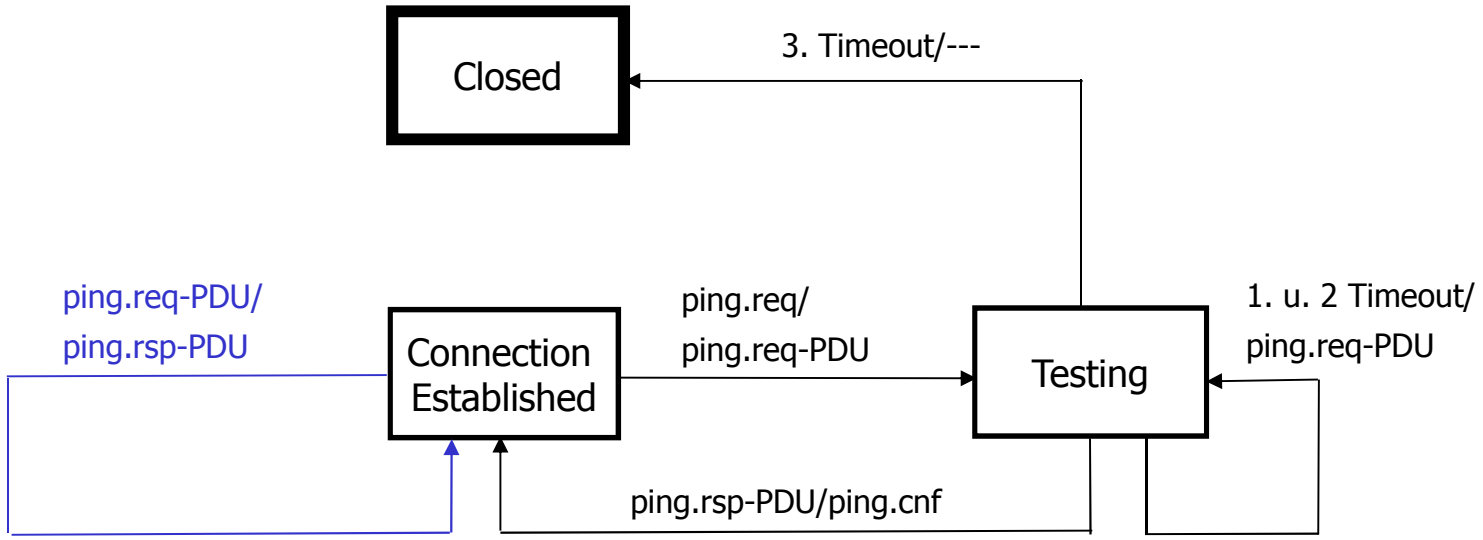
LWTRT-Spezifikation:
Spezialfall: Senden während des Verbindungsabbaus



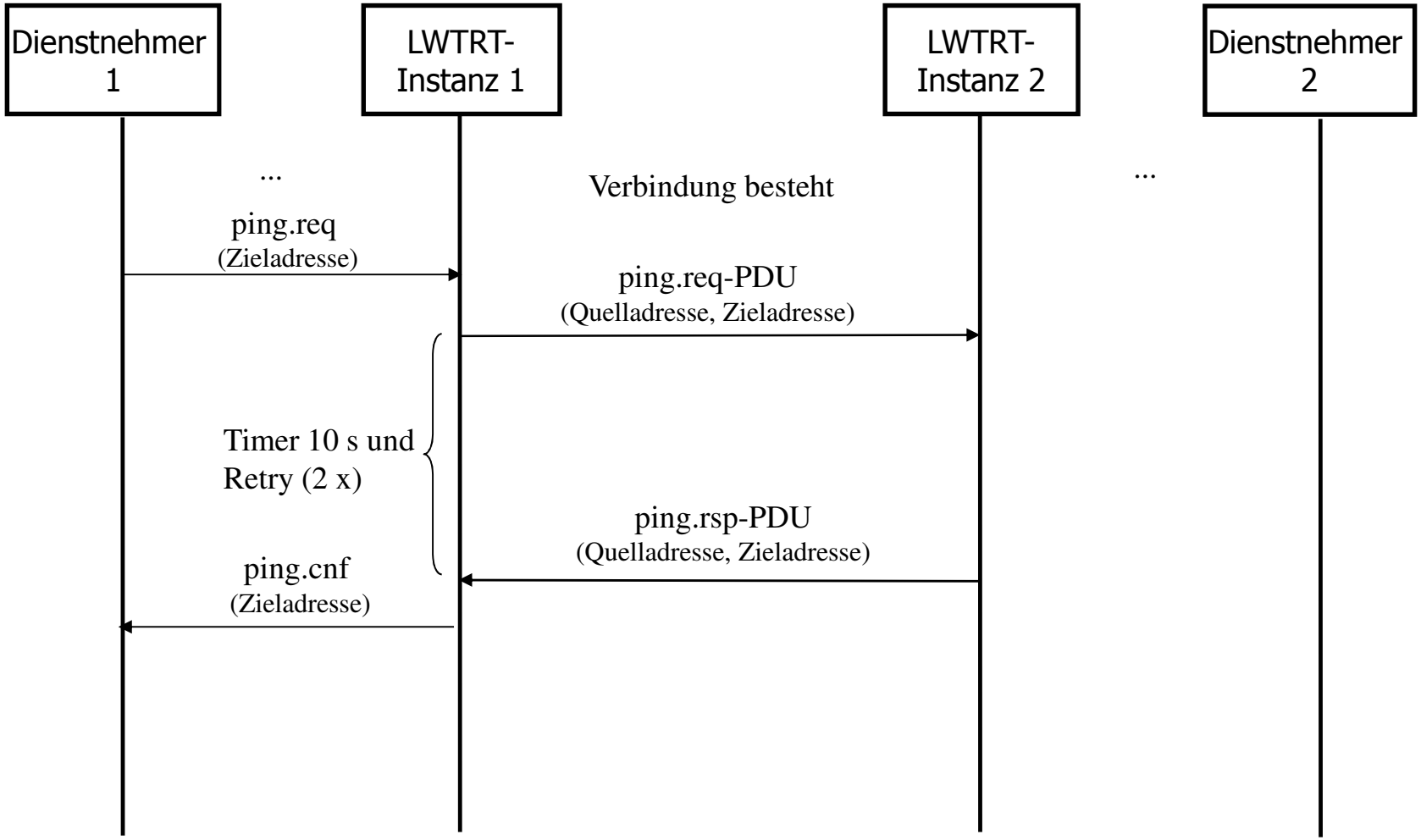
LWTRT-Zustandsautomat: Lebendüberwachung für die Verbindung

- Vereinfachte Darstellung (optional)

blau = passiv
schwarz = aktiv



LWTRT-Spezifikation: Lebendüberwachung für die Verbindung



LWTRT-Spezifikation: Nachrichten

- connect.req-PDU (1): Verbindungsanfrage
- connect.rsp-PDU (2): Verbindungsbestätigung
- disconnect.req-PDU (3): Verbindungsabbau einleiten
- disconnect.rsp-PDU (4): Verbindungsabbau bestätigen
- data.req-PDU (5): Daten senden
- data.rsp-PDU (6): Datenempfang bestätigen
- ping.req-PDU (7): Anfrage zur Lebendüberwachung
- ping-rsp-PDU (8): Antwort zur Lebendüberwachung

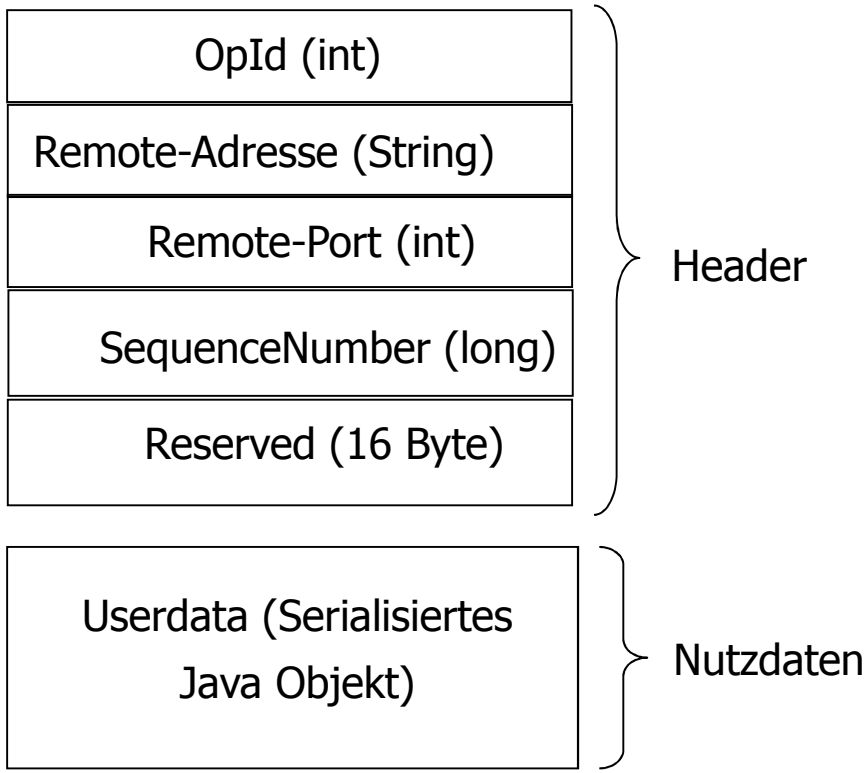
LWTRT-Spezifikation: Nachrichtenaufbau (1)

LWTRT-PDU inkl. Nutzdaten wird als Java-Objekt serialisiert:

- OpId als Integer
  - connect.req-PDU = 1, connect.rsp-PDU = 2, ..., ping.rsp-PDU = 8)
- Zieladresse (Remote-Adresse)
  - Partner-IPv4-Adresse, UDP-Portnummer des Partners
- SequenceNumber als Long
  - Folgenummer für die Nachricht
- Reserved (16 Byte)
  - Für Erweiterungen beliebig nutzbar
- UserData
  - Nutzdaten als serialisiertes Java-Objekt

LWTRT-Spezifikation: Nachrichtenaufbau (2)

- Header und Nutzdaten



LWTRT-Spezifikation: Nachrichtenaufbau – Belegung der Felder (1)

- connect.req-PDU
  - Zieladresse belegt mit IPv4-Adresse des Partner-Hosts und dem UDP-Port des Listeners von 50000 – 50999
  - SequenceNumber = 0
  - Userdata: Nicht belegt
- connect.rsp-PDU
  - Zieladresse wie Quelladresse beim connect.req
  - SequenceNumber = 0
  - Userdata: Nicht belegt

LWTRT-Spezifikation: Nachrichtenaufbau – Belegung der Felder (2)

- disconnect.req-PDU
  - Zieladresse
  - SequenceNumber
  - Userdata: Nicht belegt
- disconnect.rsp-PDU
  - Zieladresse
  - SequenceNumber
  - Userdata: Nicht belegt

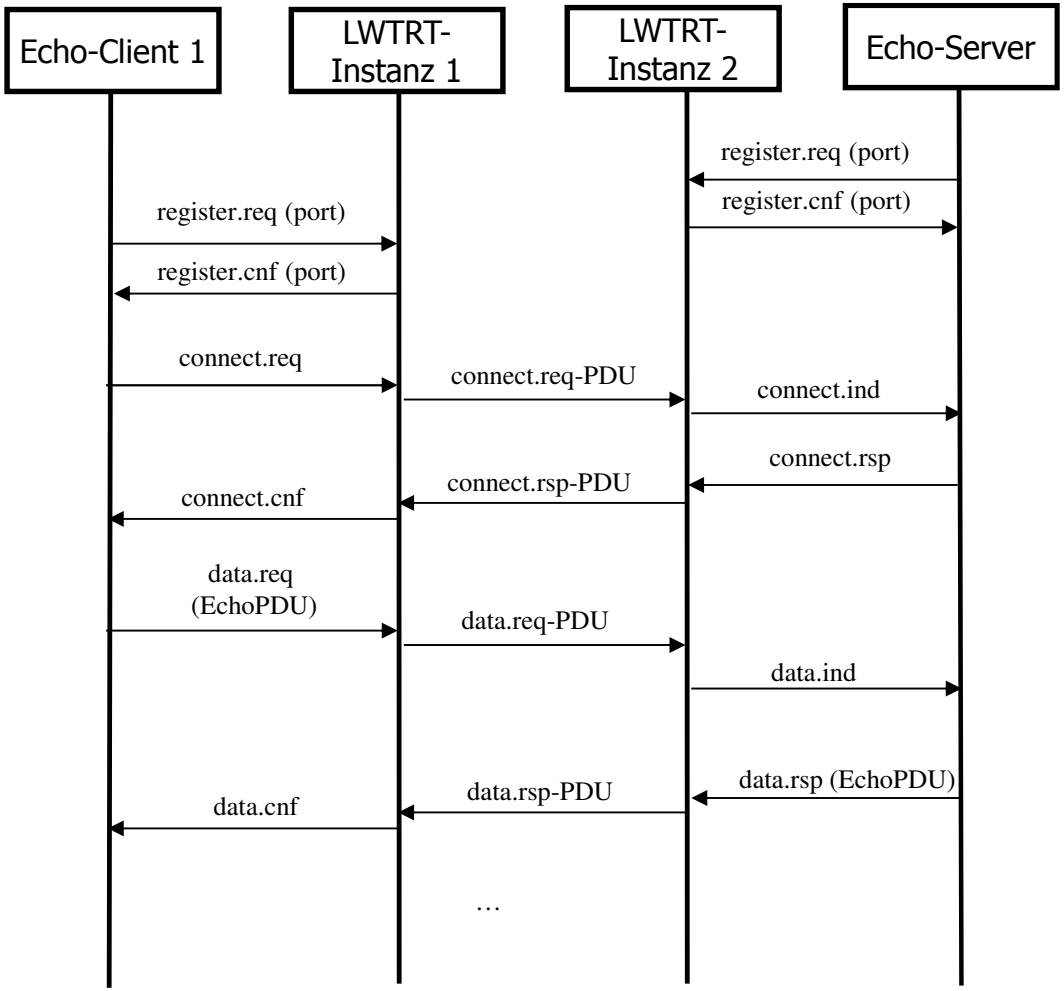
LWTRT-Spezifikation: Nachrichtenaufbau – Belegung der Felder (3)

- data.req-PDU
  - Zieladresse
  - SequenceNumber
  - Userdata: belegt
- data.rsp-PDU
  - Zieladresse
  - SequenceNumber: Wert aus korresondierender data.req-PDU
  - Userdata: nicht belegt (kein Piggybacking)

LWTRT-Spezifikation: Nachrichtenaufbau – Belegung der Felder (4)

- ping.req-PDU
  - Zieladresse
  - SequenceNumber
  - Userdata: nicht belegt
- ping.rsp-PDU
  - Zieladresse belegt
  - SequenceNumber
  - Userdata: nicht belegt

Übergreifendes Sicht:
Zusammenspiel der Protokolle beim Sessionaufbau



Überblick

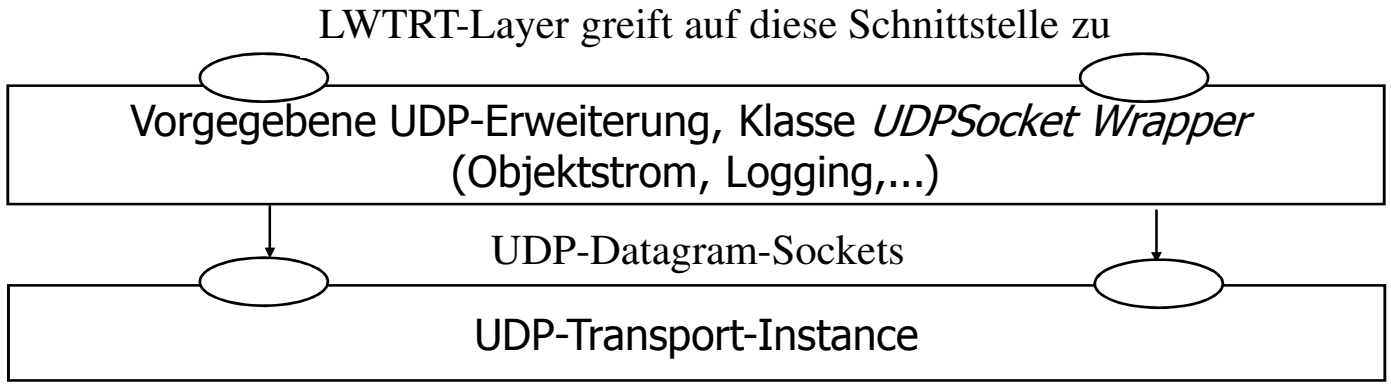
- **Einführung in die Aufgabenstellung**
  - Überblick und Lernziele
  - Teilaufgaben 1 - 7
- Details zur LWTRT-Schicht
  - Hinweise zur Protokollspezifikation
  - **Hinweise zur Implementierung**
- Sonstige Hinweise

Grundsätzliche Überlegungen

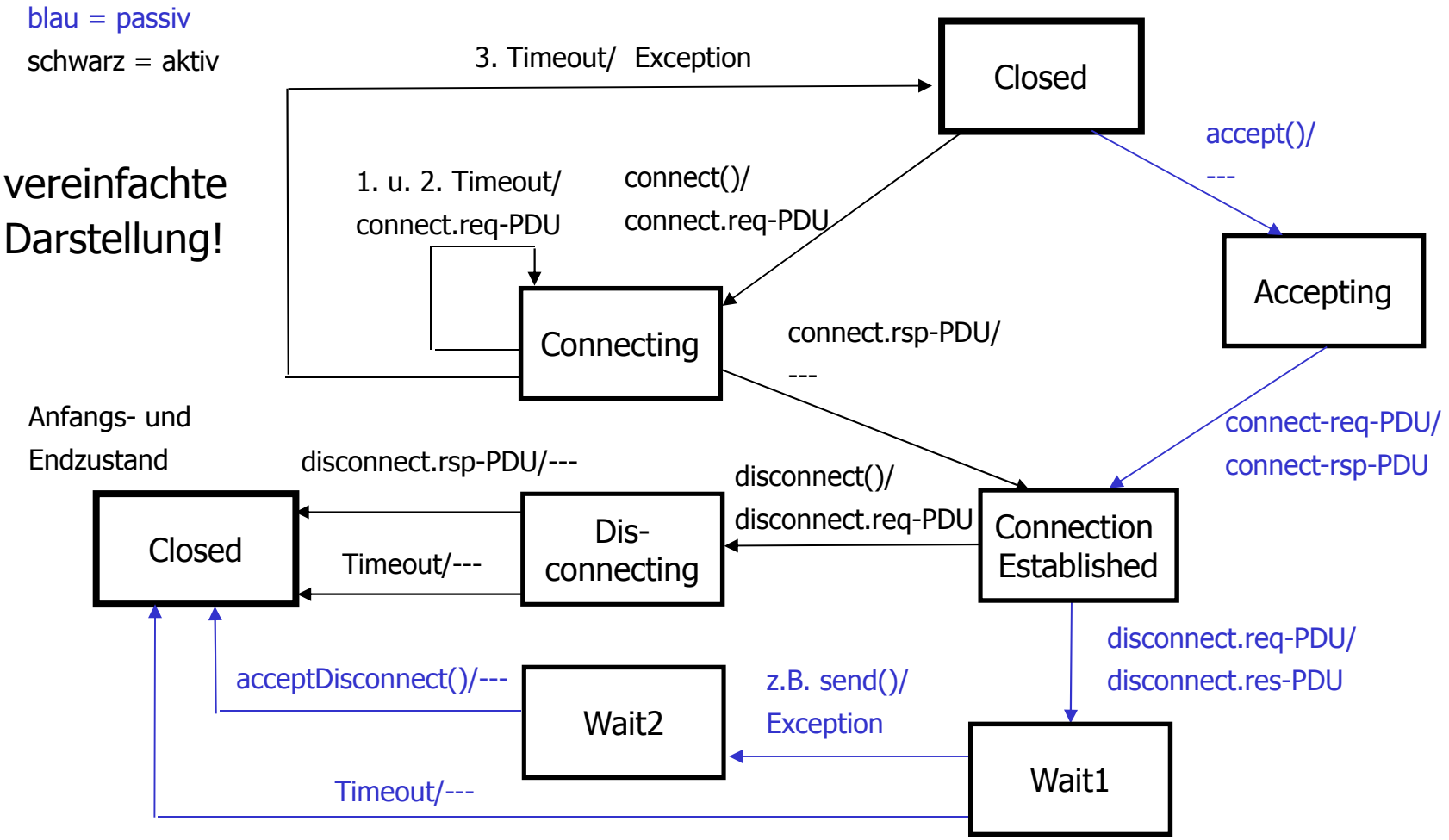
- Wie bildet man die Dienste (Dienstprimitive req, rsp, ind, cnf) auf Methoden ab?
- Wie verarbeitet man asynchron ankommende Nachrichten?
  - Ereignis muss zu einer Aktion führen
  - Callback-Mechanismus
  - Synchrone, blockierende Aufrufe
  - Wie implementiert man ein Timer-Management?
    - Timer nicht vergessen zu deaktivieren, wenn Ereignis vor Timerablauf eintritt!
  - Wie unterstützt die Programmiersprache?

UDP-Datagram-Sockets

- Es wird eine eigene Schnittstellenerweiterung bereitgestellt zu Datagram-Sockets (kein Protokoll)
- Diese erweitert die Basisimplementierung ein wenig:
  - Stellt Objektstrom auch für UDP bereit
  - Stellt Logging zur Fehlersuche bereit
  - Dient der Überprüfung der Implementierungen auf Spezifikationskonformität

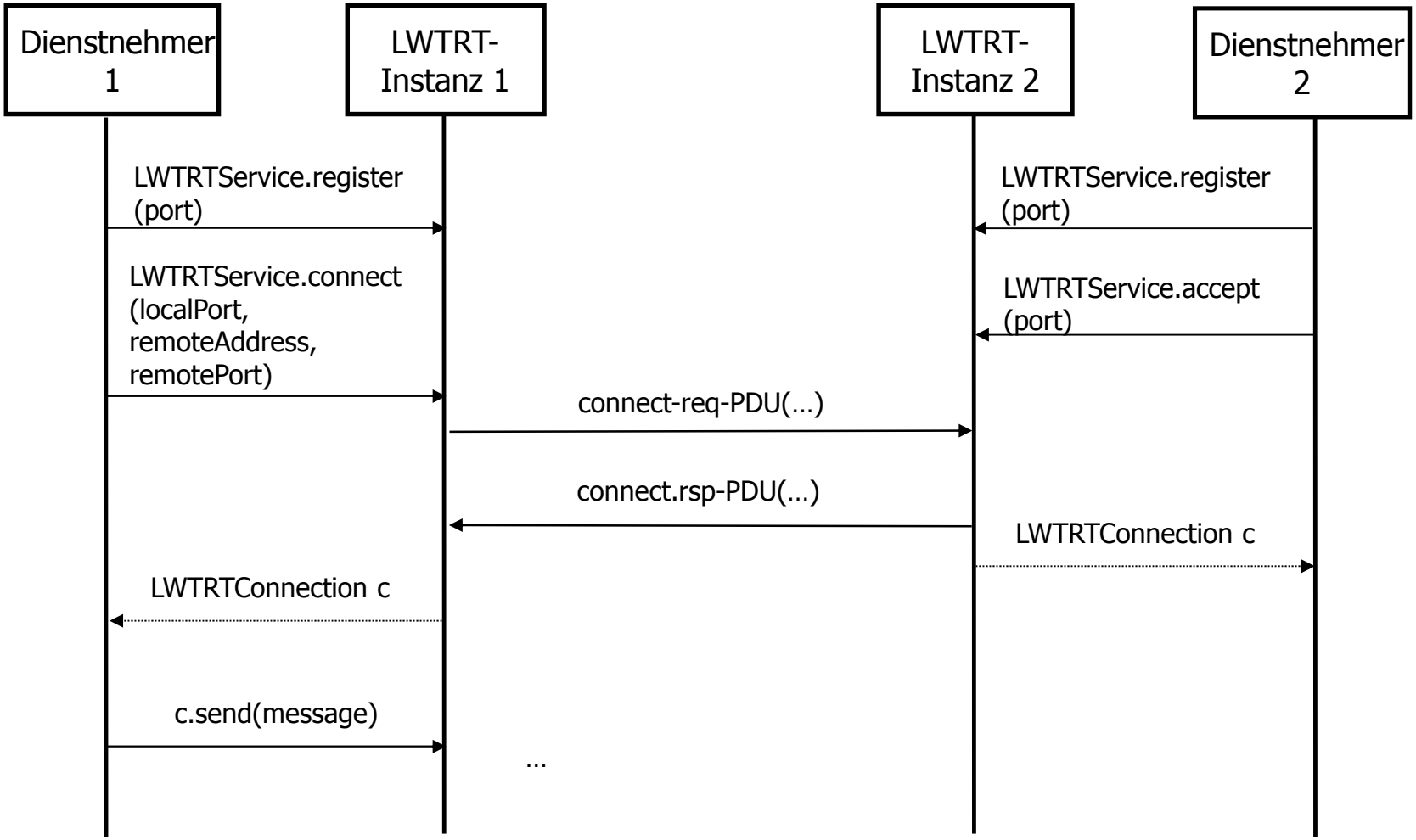


LWTRT-Implementierung in Java: LWTRT-Zustandsautomat aus Sicht des Programmierers



Je Verbindung ein Zustandsautomat bei jedem Partner

LWTRT-Implementierung in Java
Verbindungsaufbau



LWTRT-Implementierung in Java: Verbindungsaufbau

- Register-Dienst:
  - Queuelänge fix, mit max. 10 Einträgen wird erzeugt
- Connect-Dienst:
  - Synchroner Accept für Verbindungsannahme wie bei TCP
  - Connection-Object als Handle

- Diskutieren:
  - Was wird gemacht, wenn die 10 Einträge alle belegt sind, also schon 10 Clients in der Queue auf eine Verbindungsbestätigung warten?
  - Eine Möglichkeit: Keine Antwort an den Client, Client merkt dies am Timeout
  - Alternative: Connect-Rsp-PDU mit entsprechendem Fehlercode senden

LWTRT-Implementierung in Java: Java-Interfaces und Java-Klassen

- Interfaces
  - LWTRTService
  - LWTRTConnection
- Klassen
  - LWTRTPdu
- Exception-Klassen
  - LWTRTException

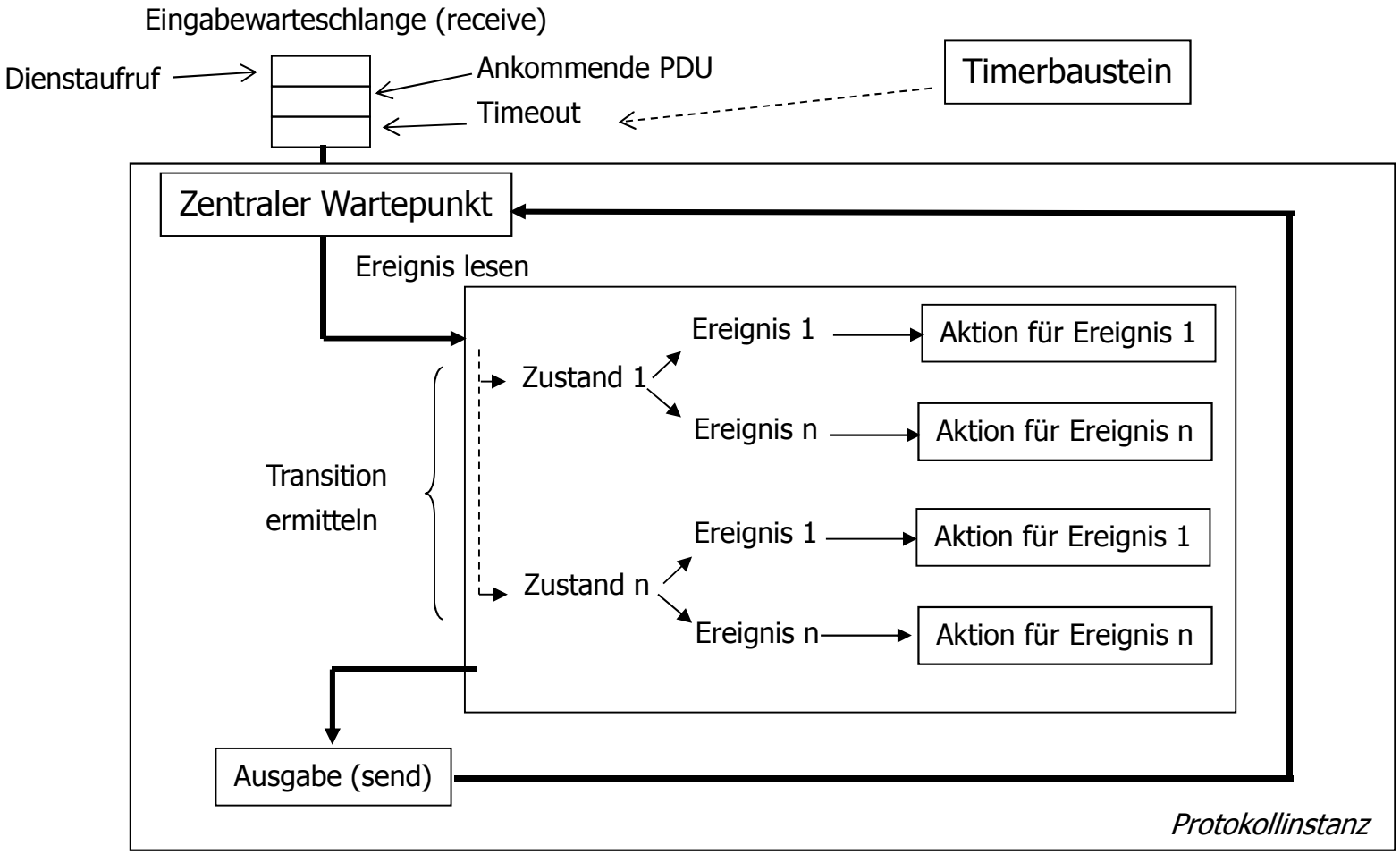
LWTRT-Referenzimplementierung in Java: Zustandsautomaten implementieren

- Wie implementiert man einen Zustandsautomaten?
  - Welches Prozess-/Threadmodell verwendet man?
    - Ein bekanntes Modell: Server-Modell
  - Auftretende Ereignisse (Nachrichten, Timeouts, Dienstaufrufe) müssen verarbeitet werden und führen zu Zustandsänderungen und Aktionen
- Mehrere Varianten
  - Bei Jedem Ereignis über Switch-Case-Konstruktionen prüfen, welche Aktion ausgeführt werden soll und ob das Ereignis zulässig ist
  - Nutzung des State Patterns zur Vermeidung von ewigen Switch-Case-Konstruktionen
- State Pattern (Objektorientierter Ansatz)
  - Jeder Zustand wird als eigene Objektklasse implementiert, das von einer Basisobjektklasse erbt

LWTRT-Referenzimplementierung in Java: Zustandsautomaten implementieren: Server-Modell

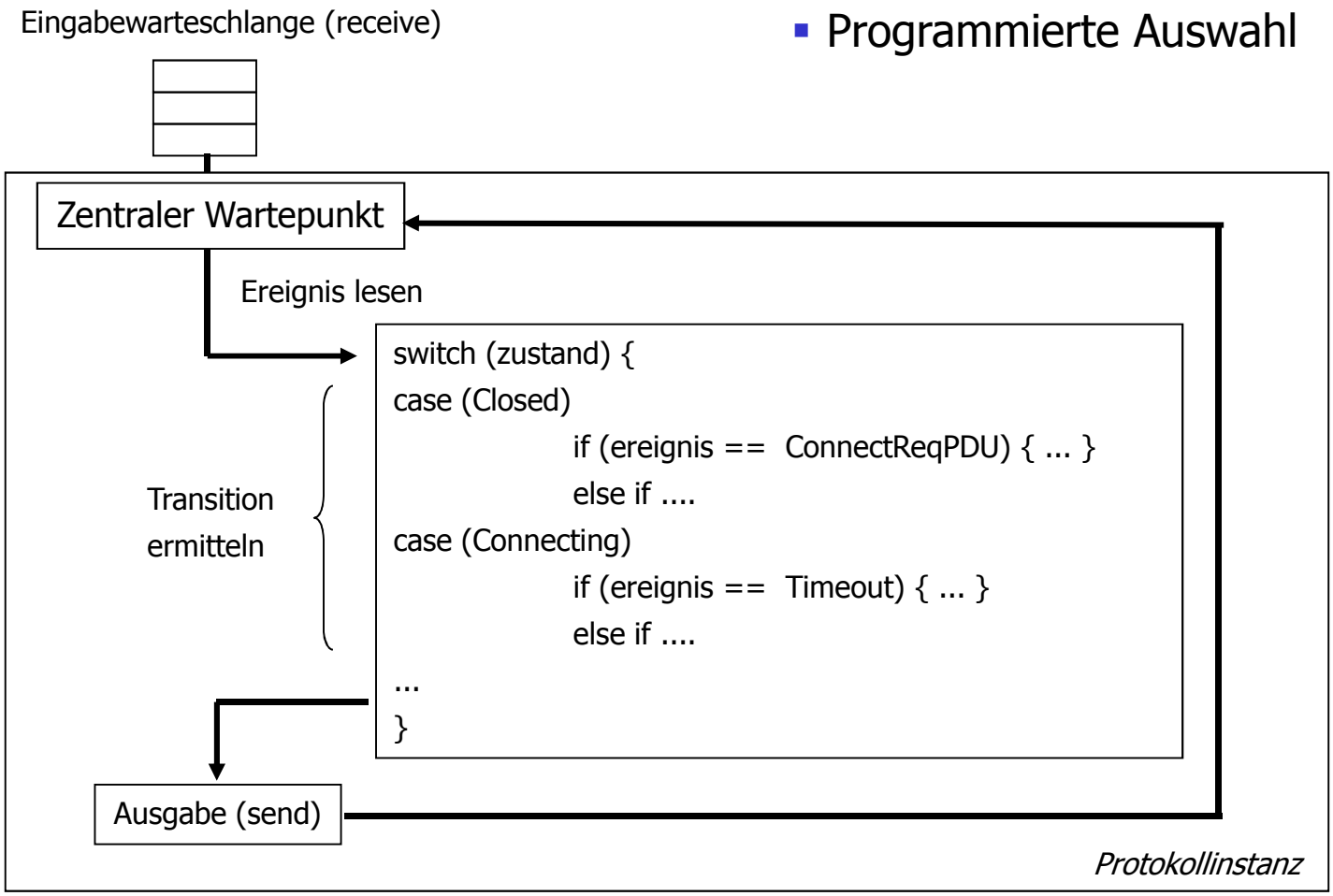
- Das Server-Modell
  - Protokollinstanz wird als zyklisch arbeitender sequentieller Prozess/Thread umgesetzt
  - Eingabewarteschlange für auftretende Ereignisse wird an einem Ereigniswartepunkt abgearbeitet
  - Ereignisse werden zyklisch in einer „Endlosschleife" abgearbeitet:
  - Grober Algorithmus:
    - Ereignis aus der Warteschlange lesen
    - Analyse des eingehenden Ereignisses (Nachricht)
    - Aktion auf Basis des aktuellen Zustandes auswählen
    - Ggf. Fehleranalyse und Fehlerreaktion
    - Aktion ausführen, Transition durchführen (in Zustand gehen)und Ausgabe erzeugen
    - Und dann wieder von vorne: Nächstes Ereignis aus der Warteschlange lesen

LWTRT-Referenzimplementierung in Java: Zustandsautomaten implementieren: Server-Modell



König, H.: Protocol Engineering, Teubner, 2003

LWTRT-Referenzimplementierung in Java: Zustandsautomaten implementieren: Server-Modell

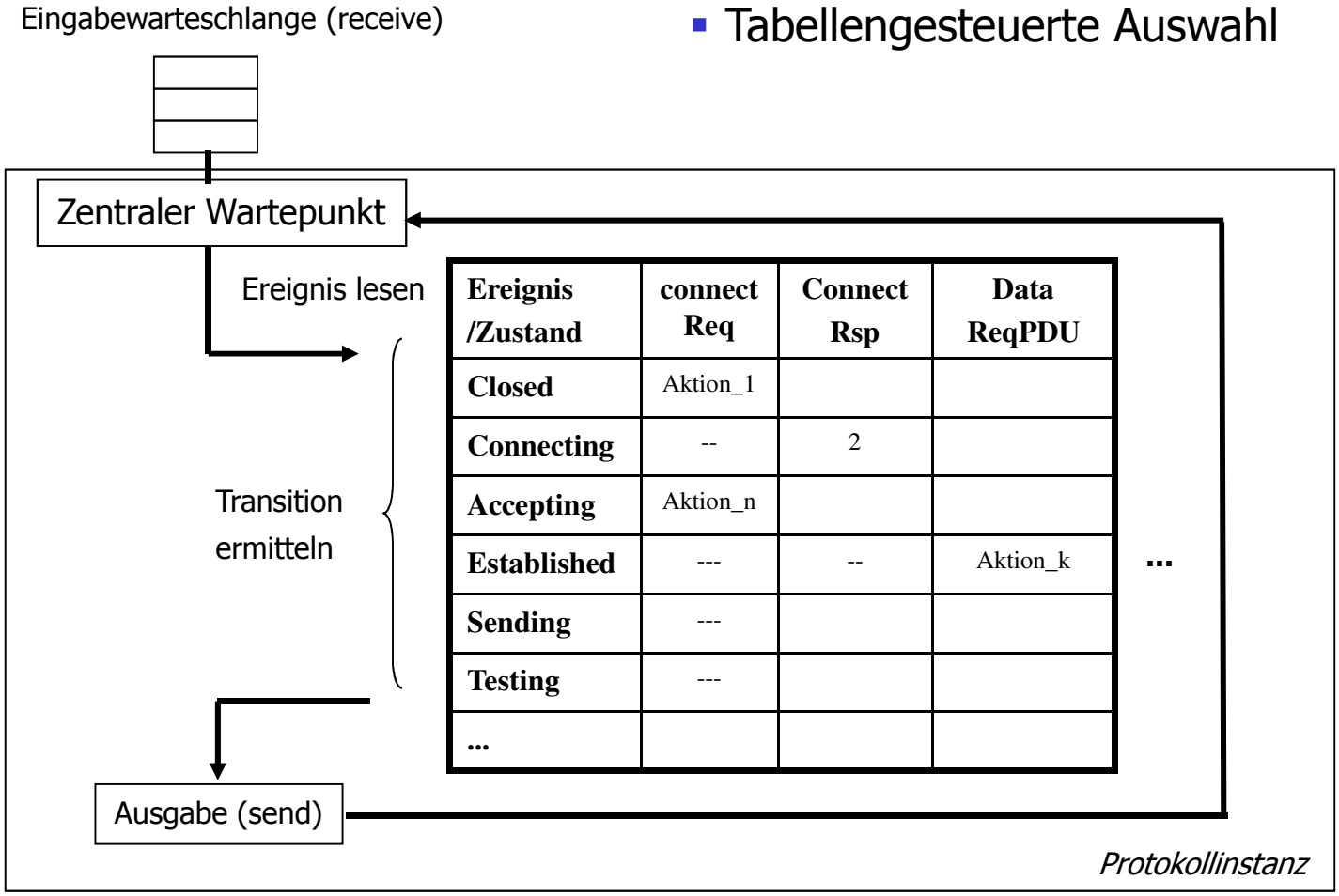


- Programmierte Auswahl

König, H.: Protocol Engineering, Teubner, 2003

LWTRT-Referenzimplementierung in Java: Zustandsautomaten implementieren: Server-Modell

- Tabellengesteuerte Auswahl



Ereignis /Zustand	connect Req	Connect Rsp	Data ReqPDU
Closed	Aktion_1		
Connecting	--	2	
Accepting	Aktion_n		
Established	---	--	Aktion_k
Sending	---		
Testing	---		
...			

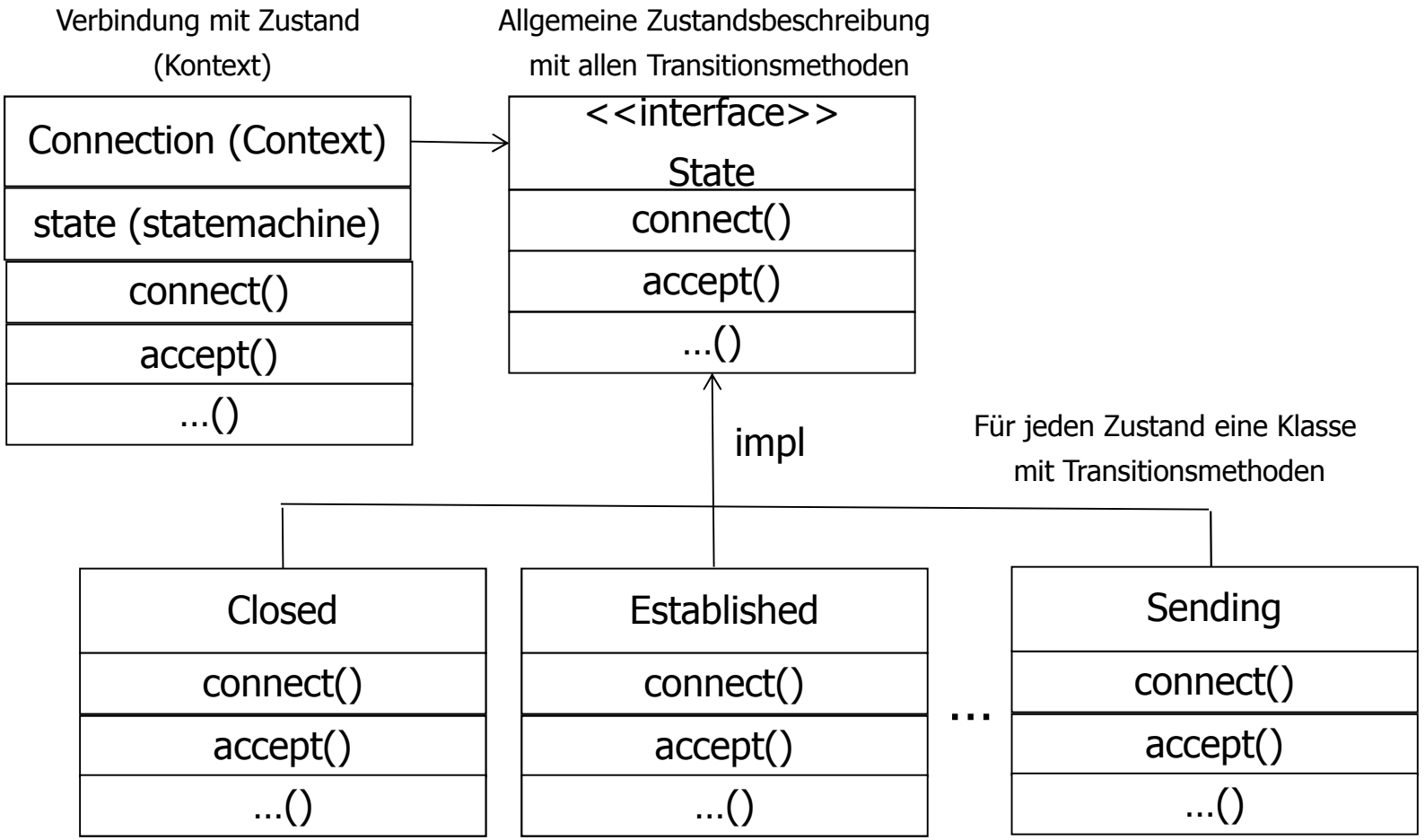
...

König, H.: Protocol Engineering, Teubner, 2003

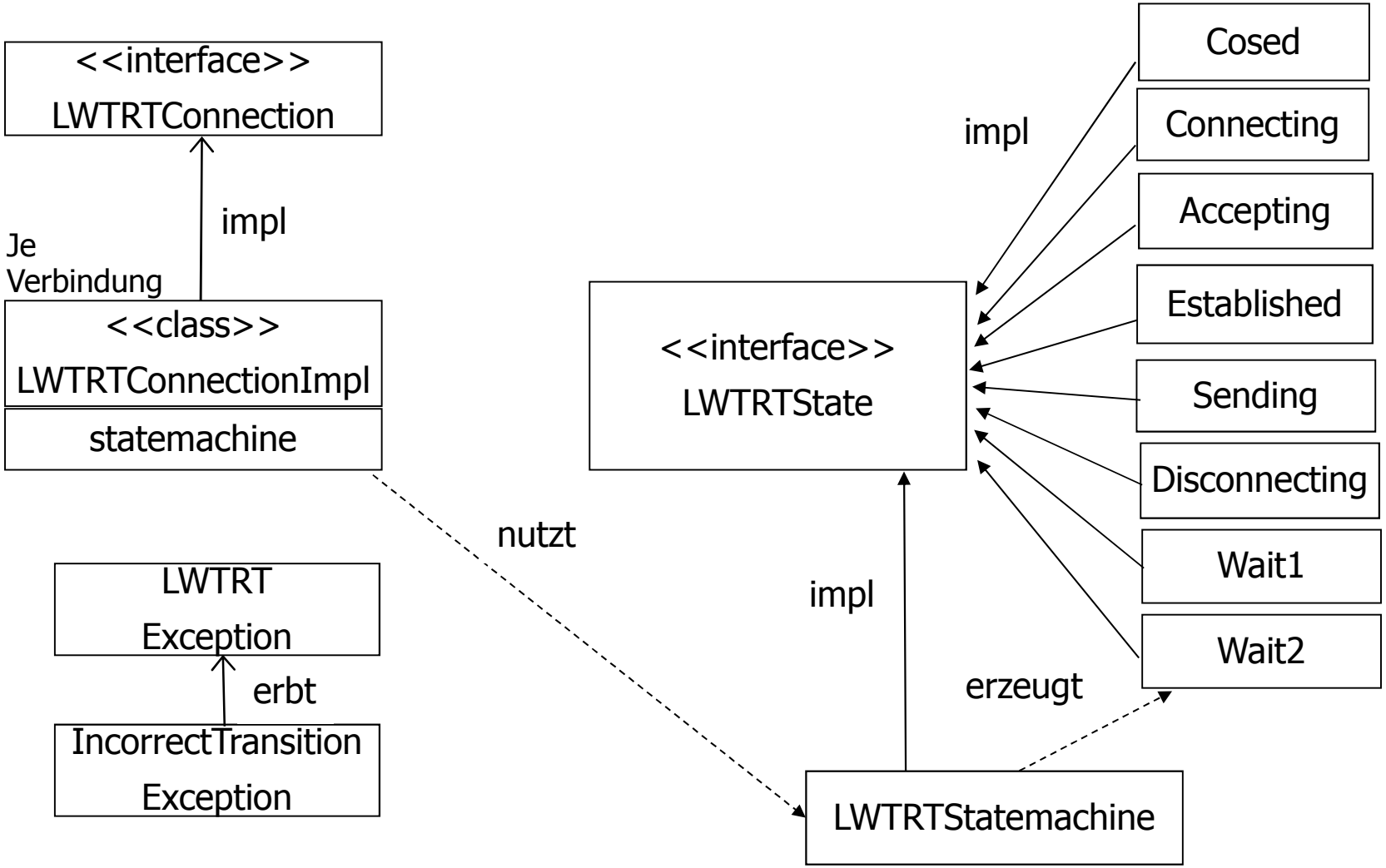
LWTRT-Referenzimplementierung in Java: Zustandsautomaten implementieren: Server-Modell

- Kritik am Server-Modell
  - Positiv
    - Einfacher Entwurf, da Zustandsautomat direkt abgebildet wird
    - Einfach zu implementieren
  - Negativ
    - Bei vielen Zuständen unübersichtlich, da viele Switch-Case und If-Konstruktionen zur Ermittlung der nächsten Transition notwendig sind
    - Evtl. nicht so leistungsfähig, da alles in einem Prozess/Thread erledigt wird
      - → Verbesserung durch Ausführen einer Aktion in einem eigenen Thread

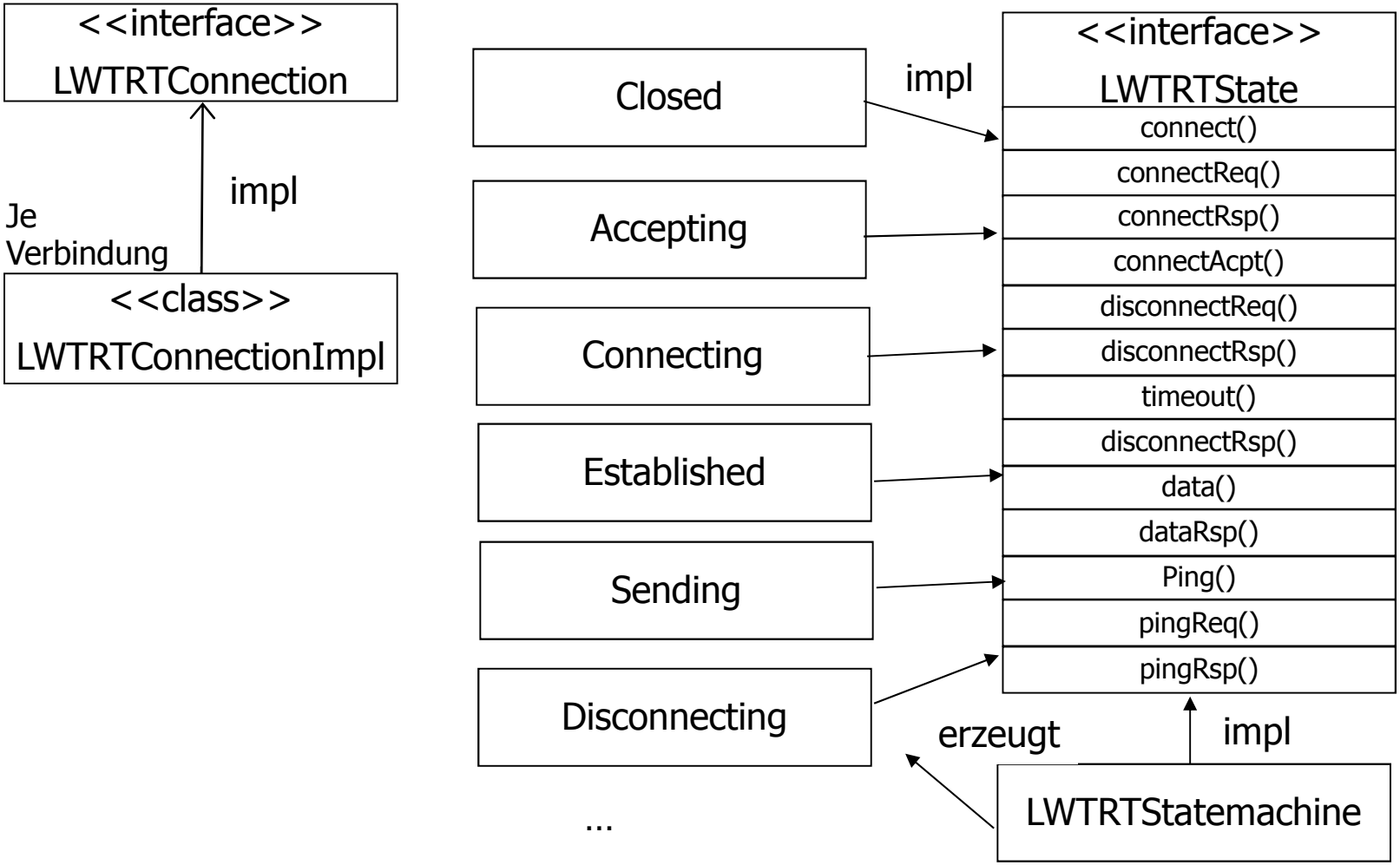
LWTRT-Referenzimplementierung in Java: State Pattern für Protokollautomaten (allgemein)



LWTRT-Referenzimplementierung in Java: State Pattern konkret für LWTRT



LWTRT-Referenzimplementierung in Java: Zustandsautomat objektorientiert mit State Pattern



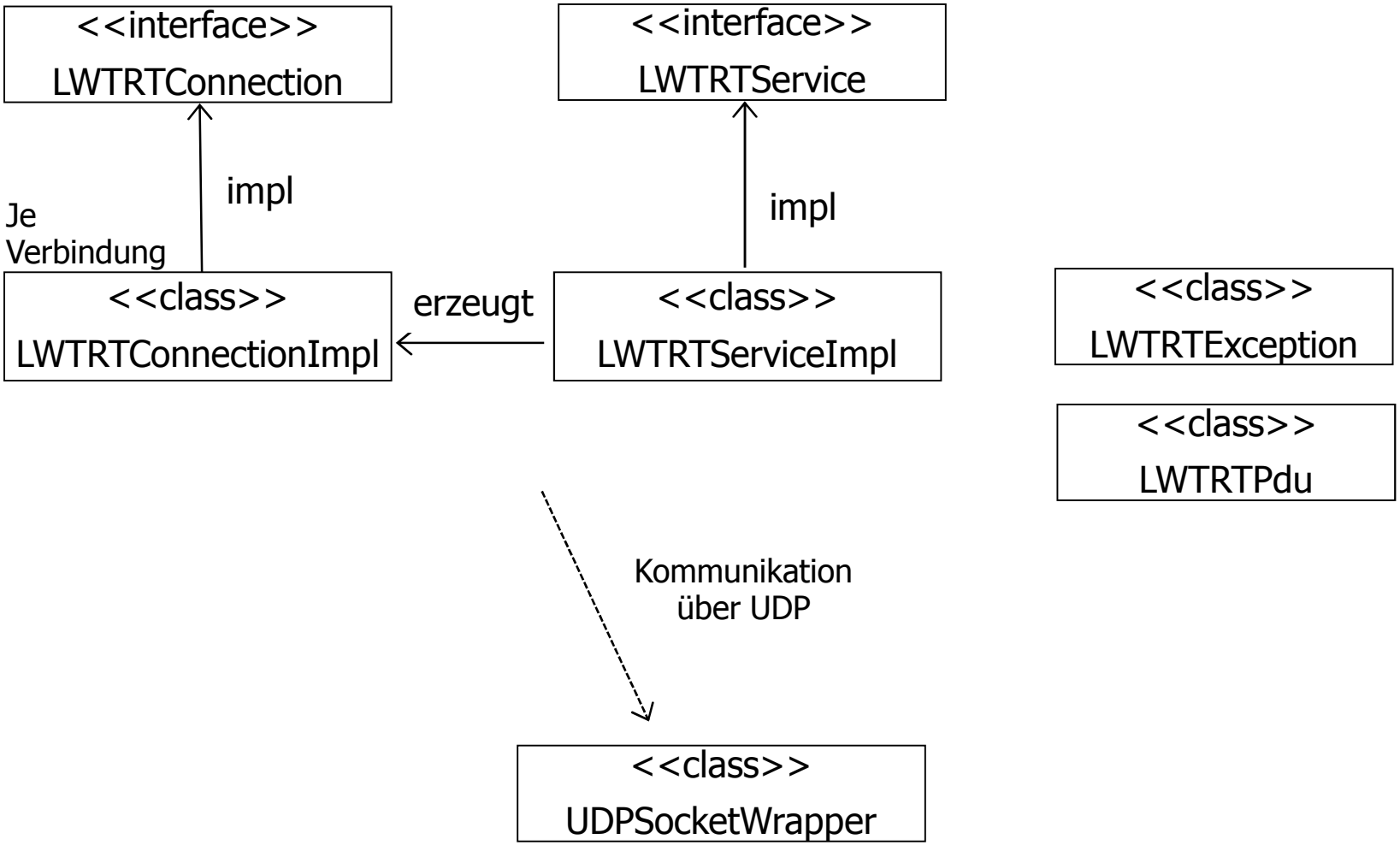
LWTRT-Referenzimplementierung in Java: Zustandsautomaten implementieren: Sonstiges (1)

- Zeitüberwachung programmieren
  - In Java:
    - System.currentTimeMillis() oder System.nanoTime()
    - Warten mit Thread.sleep()
- Kodierung der PDUs
  - In Java über Objektserialisierung (schon vorgegeben)
- Jedes Ereignis (ankommende PDU, Dienstaufruf) ist zu überprüfen
  - Ist das Ereignis im aktuellen Zustand erlaubt?
  - Wie reagiert man auf fehlerhafte Ereignisse?
    - Exception nach oben weiterleiten (Dienstnutzer soll entscheiden)
    - Abbruch nur, wenn schwerwiegende Systemfehler
    - Ignorieren, wenn nicht schwerwiegend
    - Immer einen Logsatz schreiben

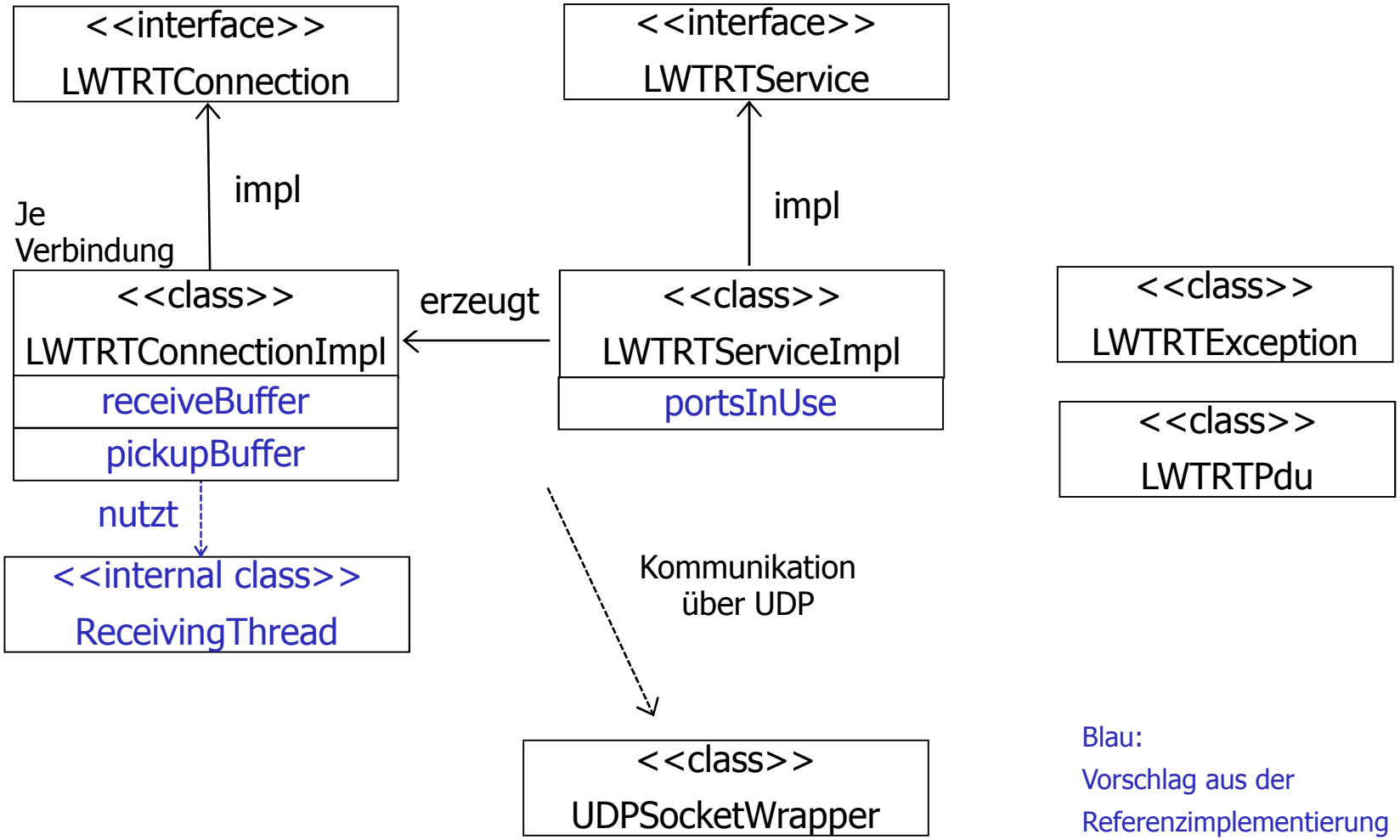
LWTRT-Referenzimplementierung in Java: Zustandsautomaten implementieren: Sonstiges (2)

- Verbindungsverwaltung
  - Verbindungstabelle führen
    - Adresse des Partners
    - Zustand
- Zwischenpufferung von ankommenden PDUs
  - Entkoppelung des Empfangsvorgangs von der sonstigen Verarbeitung zur Vermeidung von Blockierungen
- Generell: Logging / Debugging in getrennte Logdateien
  - Logsatz schreiben (Typen: INFO, DEBUG, ERROR) bei
    - Ankommenden PDUs nach dem Empfang (Ereignis als DEBUG, Inhalt als INFO)
    - Abgehende PDUs, vor dem Senden (Ereignis als DEBUG, Inhalt als INFO)
    - Methodenaufrufen (DEBUG)
    - Fehlersituationen (ERROR)

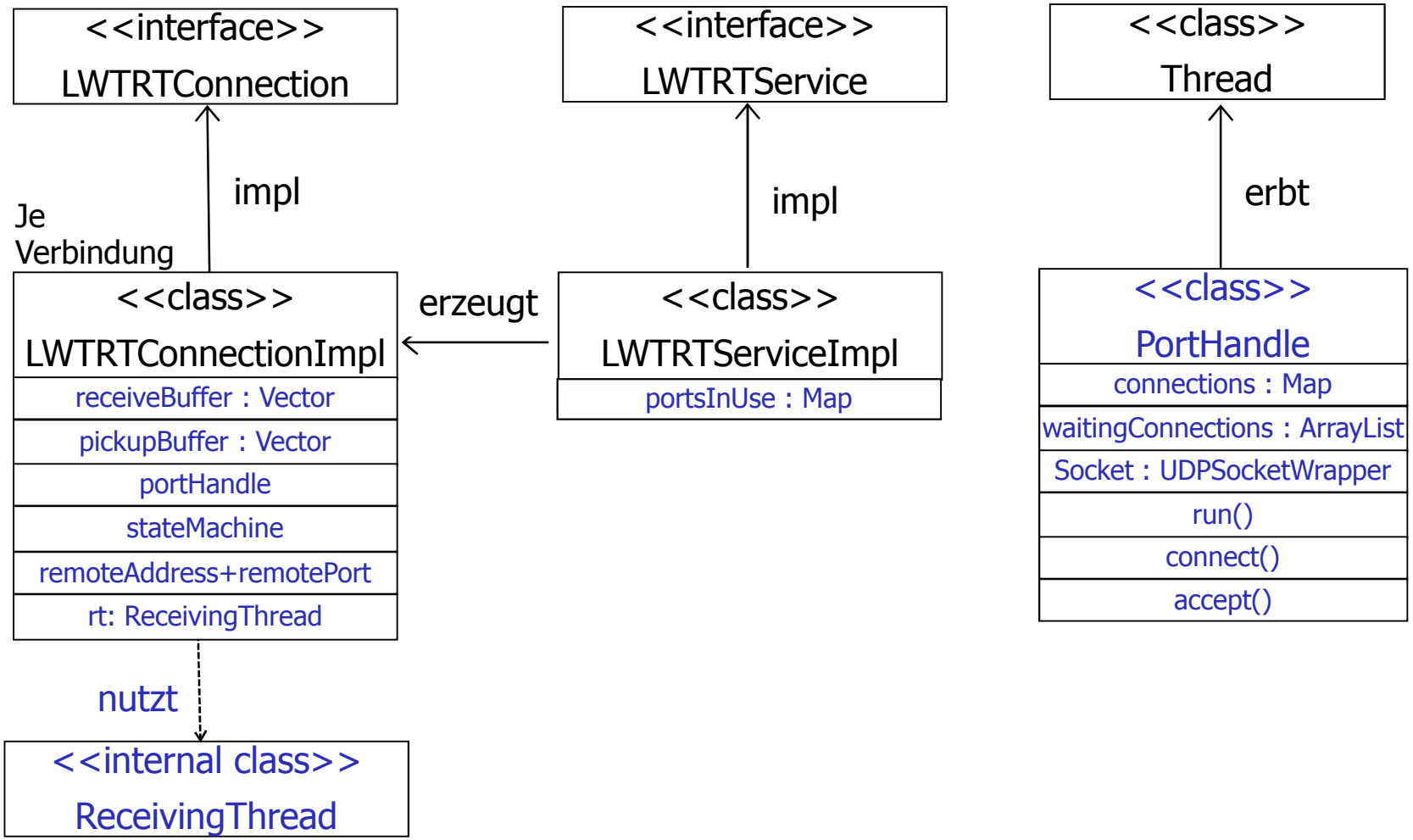
LWTRT-Referenzimplementierung in Java: Java-Interfaces und Java-Klassen



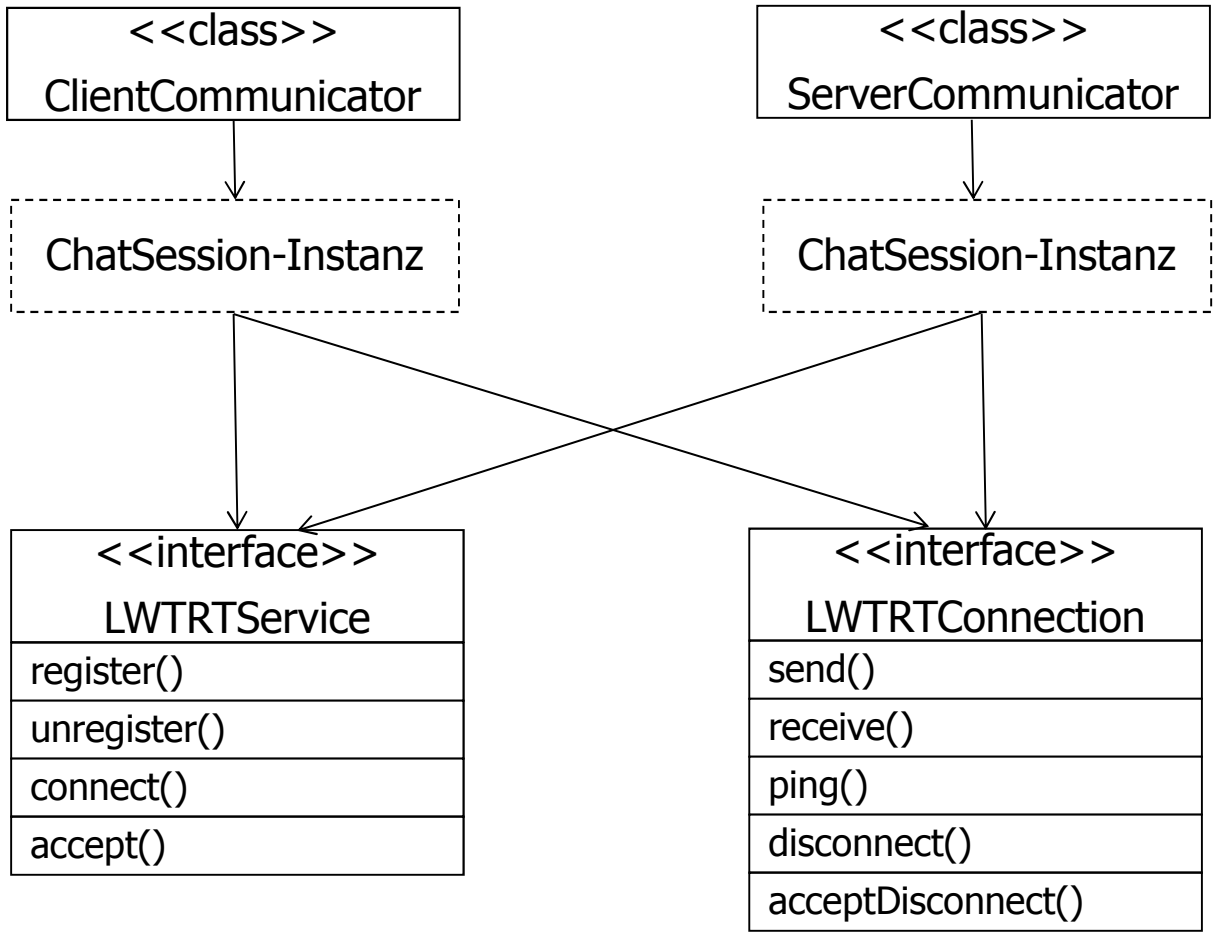
LWTRT-Referenzimplementierung in Java: Java-Interfaces und Java-Klassen



LWTRT-Referenzimplementierung in Java: Java-Interfaces und Java-Klassen



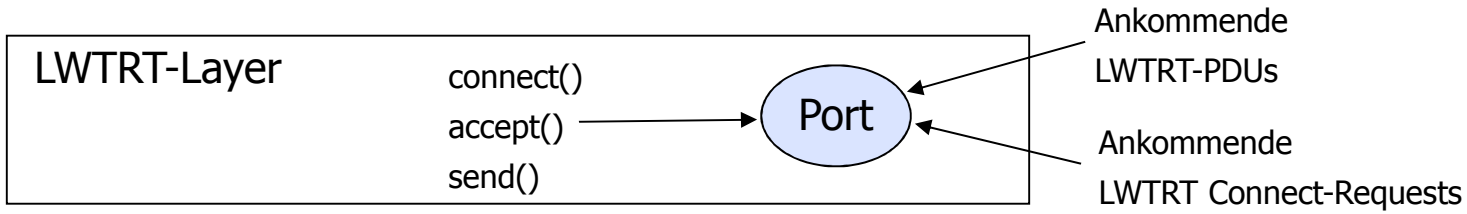
LWTRT-Referenzimplementierung in Java: LWTRTService und LWTRTConnection



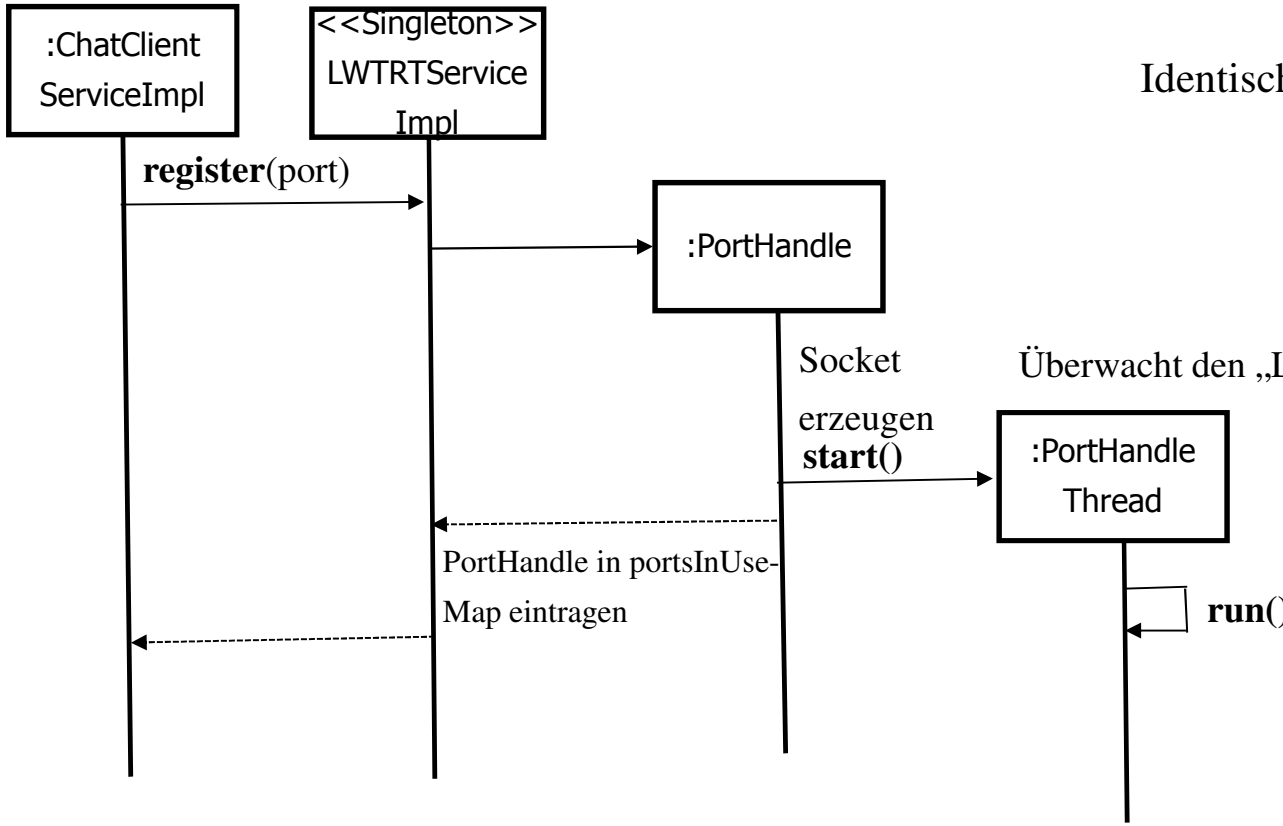
LWTRT-Referenzimplementierung in Java: Der registrierte Port

- Über den registrierten UDP-Port läuft die **gesamte** Kommunikation
  - Ankommende Nutzdatennachrichten werden entgegengenommen
  - Abgehende Nachrichten werden über den Port gesendet
  - Verbindungsaufbauwünsche beliebiger Partner können ankommen, LWTRT erzeugt gleich eine Verbindung und weist nichts ab

    → einfache Lösung, normalerweise müsste Anwendung entscheiden!
- Verbindungsaufbauwünsche (connect) der lokalen Anwendung werden über den Port bearbeitet
- Das Warten auf ankommende Verbindungsaufbauwünsche (accept) wird über den Port realisiert
- Die ankommenden Nutzdaten dürfen nicht interpretiert werden!



LWTRT-Referenzimplementierung in Java: Registrierung



Identisch in Client und Server

Überwacht den „Listen“-Port

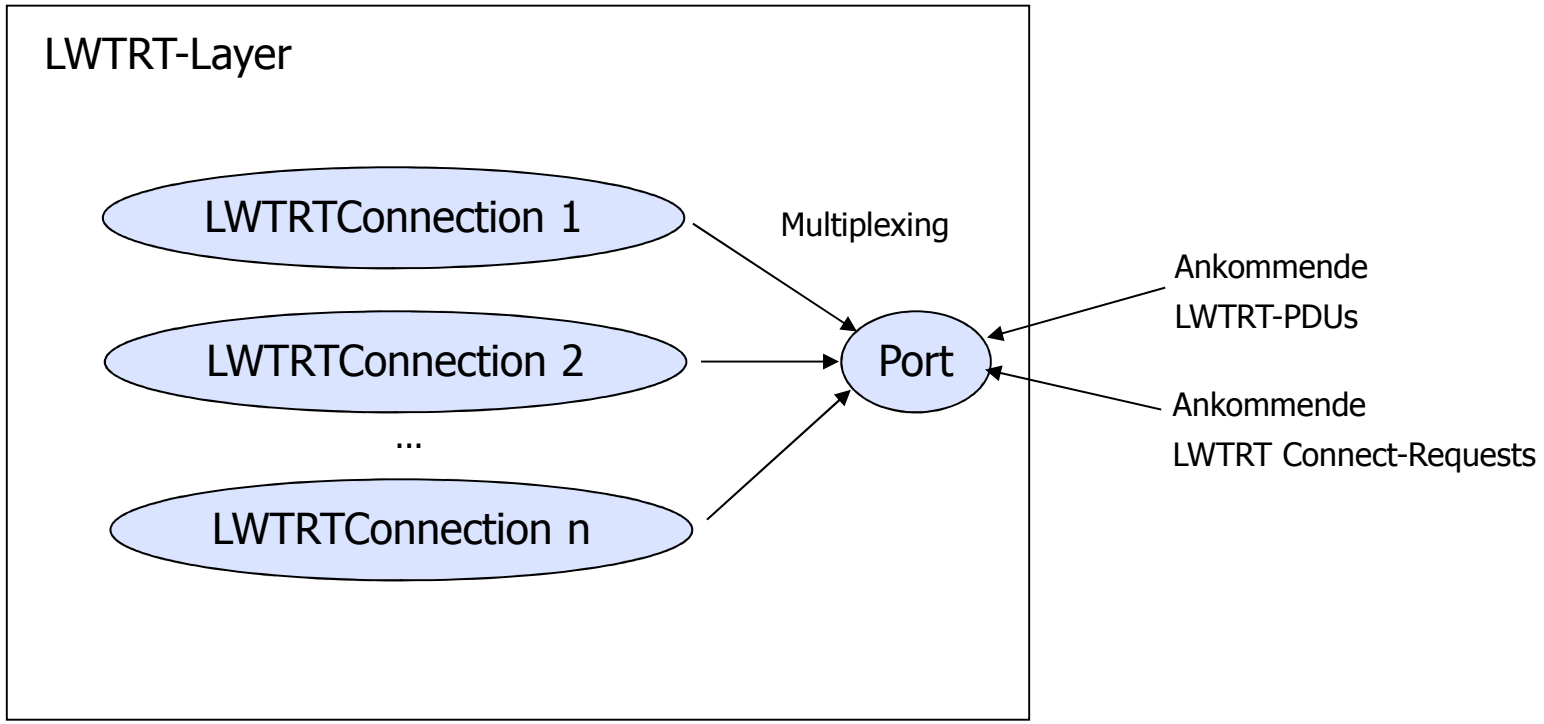
```
while (!stop) {
    socket.receive(lwtrtPdu);
    PDU analysieren
    if (bestehende Verbindung) {
        PDU in receiveBuffer legen
    } else {
        Neue Verbindung annehmen
    }
}
```

Bisher noch keine Verbindung erzeugt,

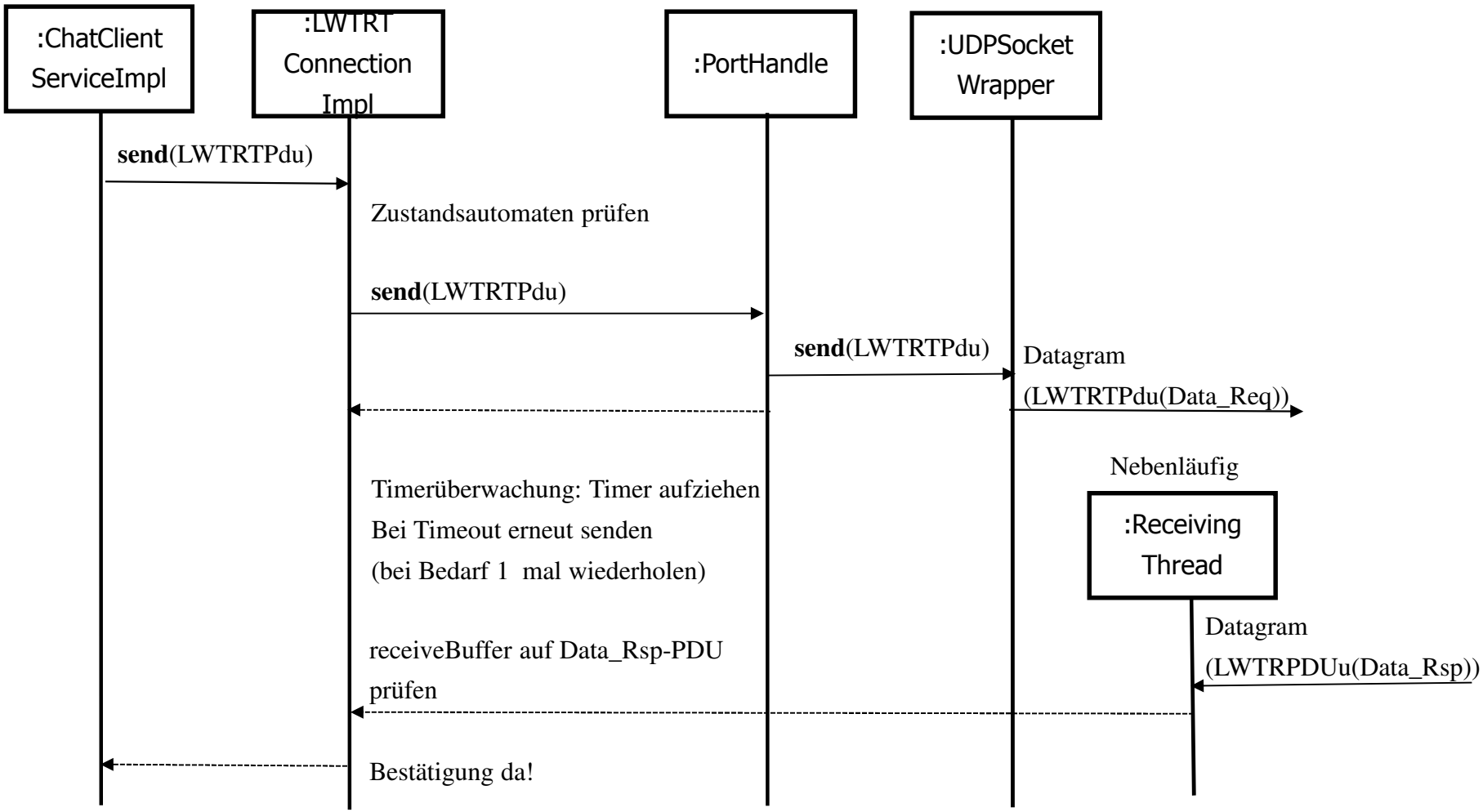
nur „Listenport“ in PortHandle aktiviert!

LWTRT-Referenzimplementierung in Java: Multiplexing

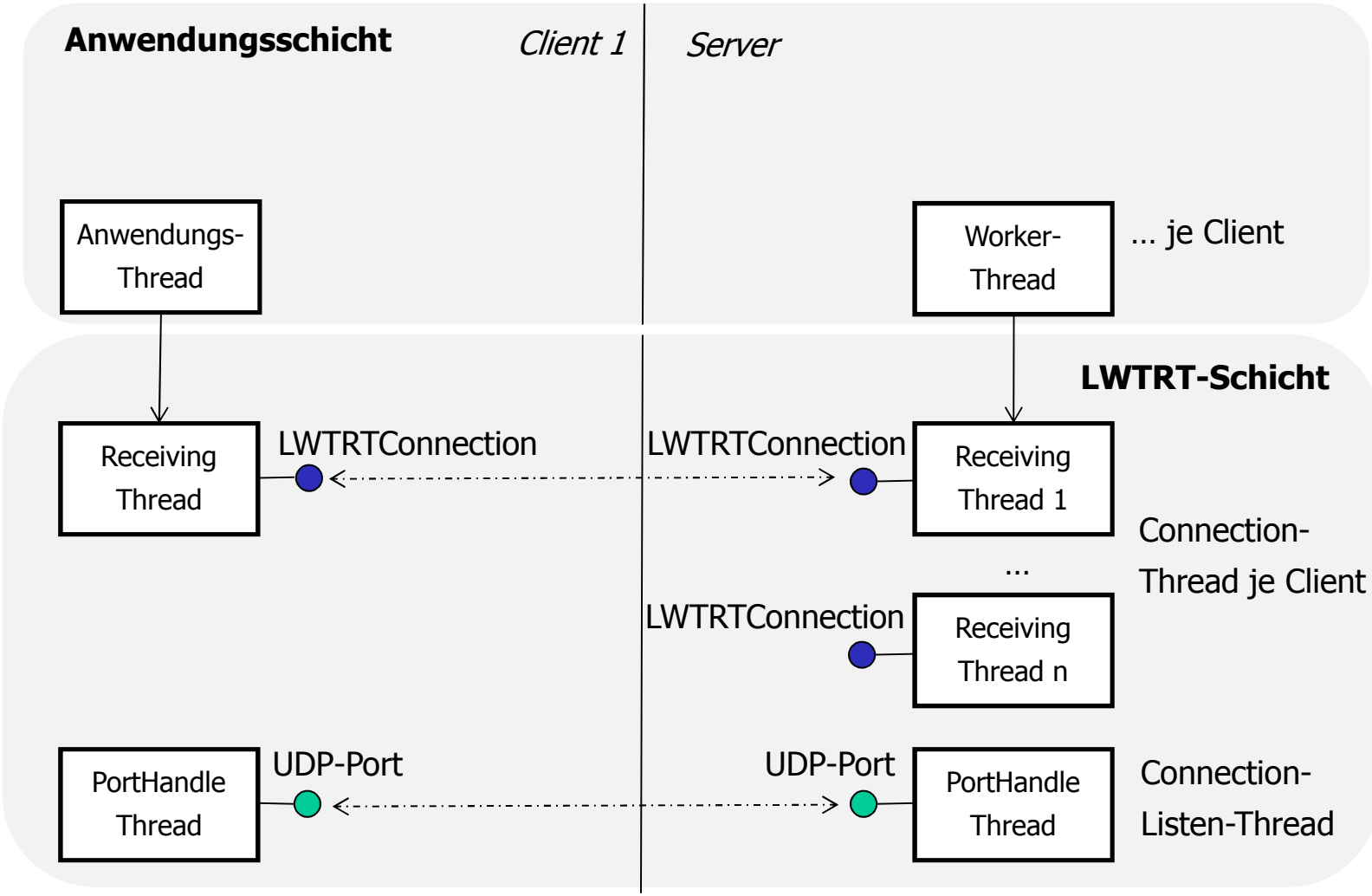
- Alle Verbindungen werden über einen UDP-Port (den registrierten Port) „gemultiplexed“



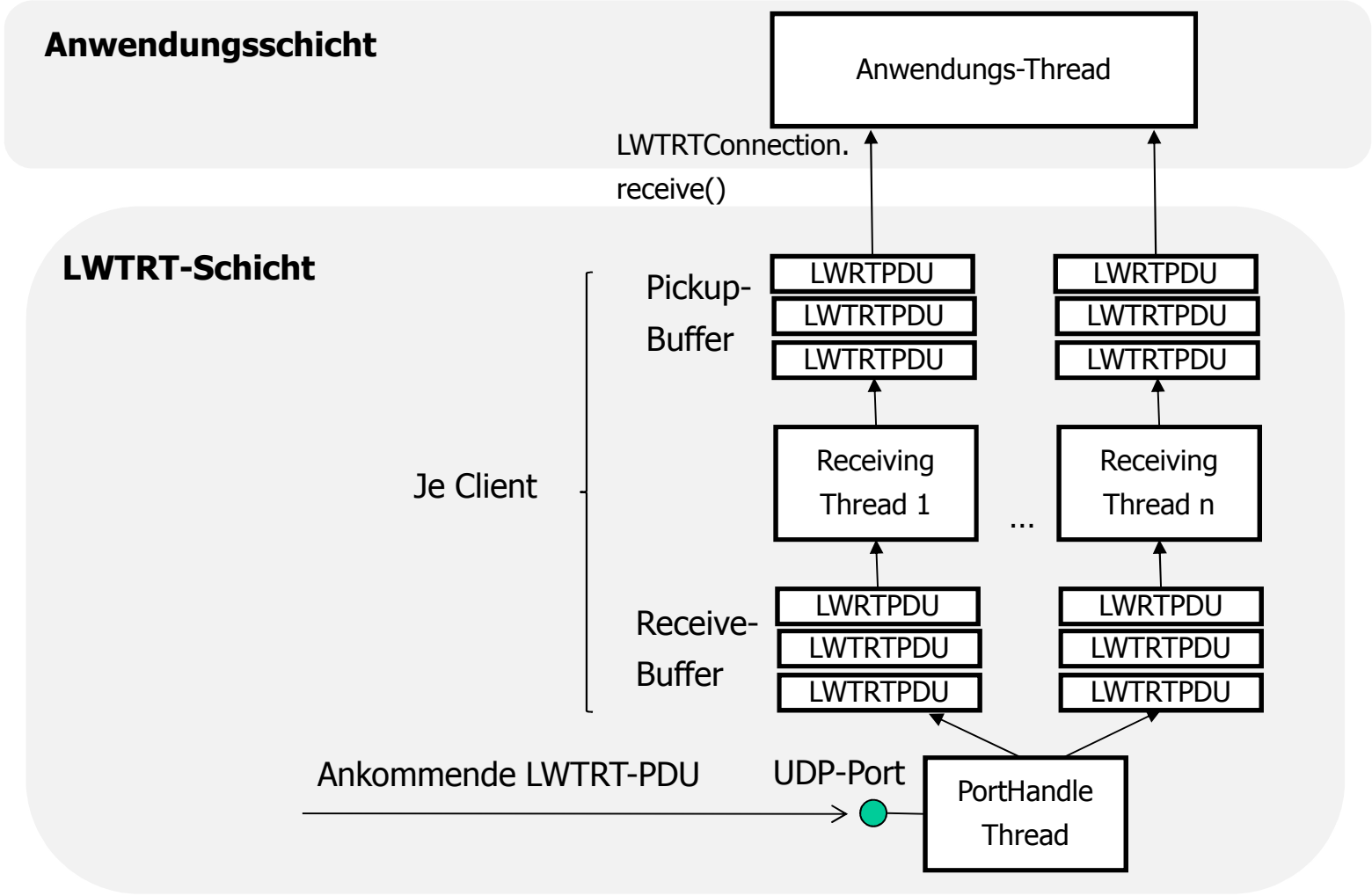
LWTRT-Referenzimplementierung in Java: Aktives Senden



Referenzimplementierung in Java: Threadmodell



Referenzimplementierung in Java: Nachrichtenverarbeitung, serverseitig



Rückblick

1. Einführung in die Aufgabenstellung
   - Überblick und Lernziele
   - Teilaufgaben 1 - 7
2. Details zur LWTRT-Schicht
   - Hinweise zur Protokollspezifikation
   - Hinweise zur Implementierung