Design und Implementierung eines Peer-to-Peer-basierten Overlaynetzwerkes

Kolloquium zur Bachelorarbeit

von Nils Rohwedder

31.01.2012

UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR TELEMATIK

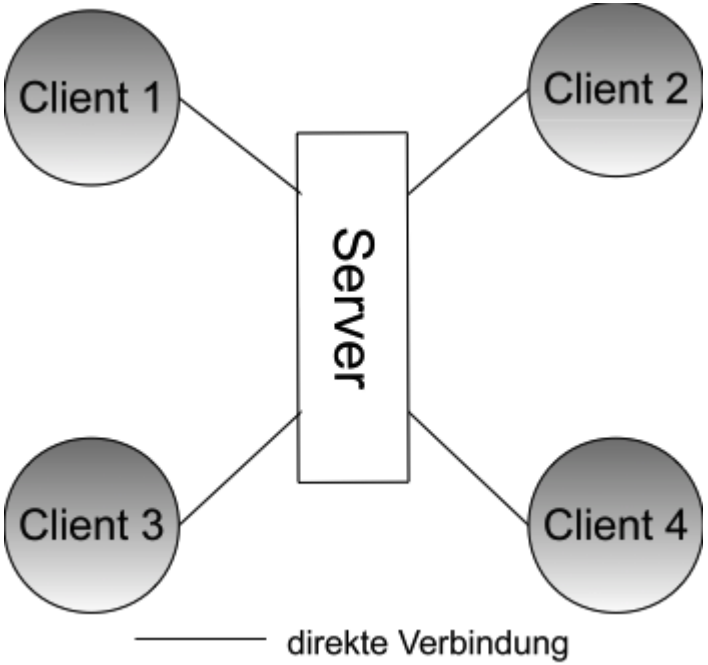
Motivation

- WISEBED
  - Entwickelt im Rahmen der europäischen FIRE-Initiative
  - *Testbed-Runtime*
    - Erlaubt den Betrieb einer weltweiten Testumgebung für Algorithmen, Protokolle und Anwendungen für drahtlose Sensornetzwerke
    - Arbeitet verteilt auf Komponenten der PC-, Server- oder Netbook-Klasse
    - Verwendet zur Kommunikation zwischen den Komponenten ein selbstentwickeltes Overlaynetzwerk

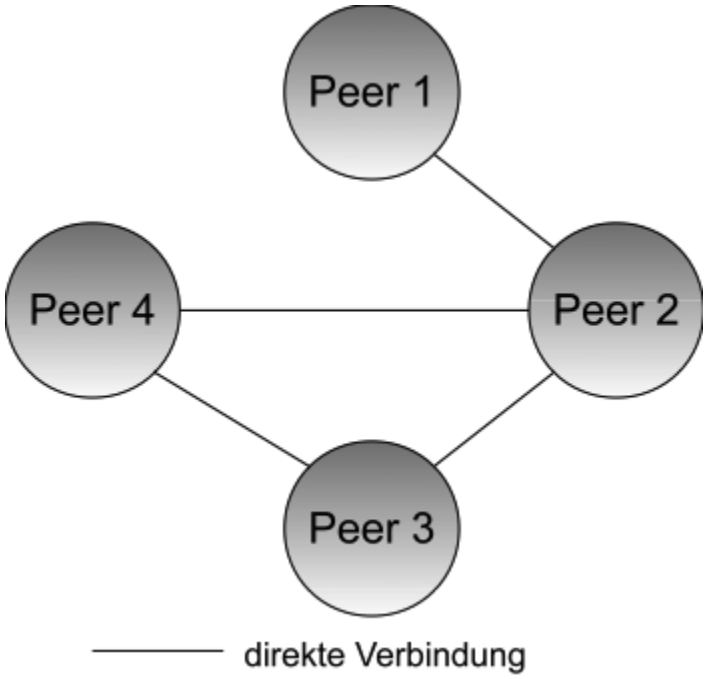
Aufgabenstellung & Herangehensweise

- Middleware für ein Peer-to-Peer Overlaynetzwerk
- Ziele:
  - Ersetzung des Overlaynetzwerks von *Testbed-Runtime*
  - Einsetzbarkeit in anderen verteilten Anwendungen durch Verwendung von *Message Exchange Patterns (Unicast, RPC)*
  - Evaluation der Funktionalität und ausreichenden Performanz
- Herangehensweise:
  - Identifizierung und Beschreibung der Message Exchange Pattern
  - Entwurf des Protokolls und anschließende Implementierung
  - Evaluation

Client-Server und Peer-to-Peer

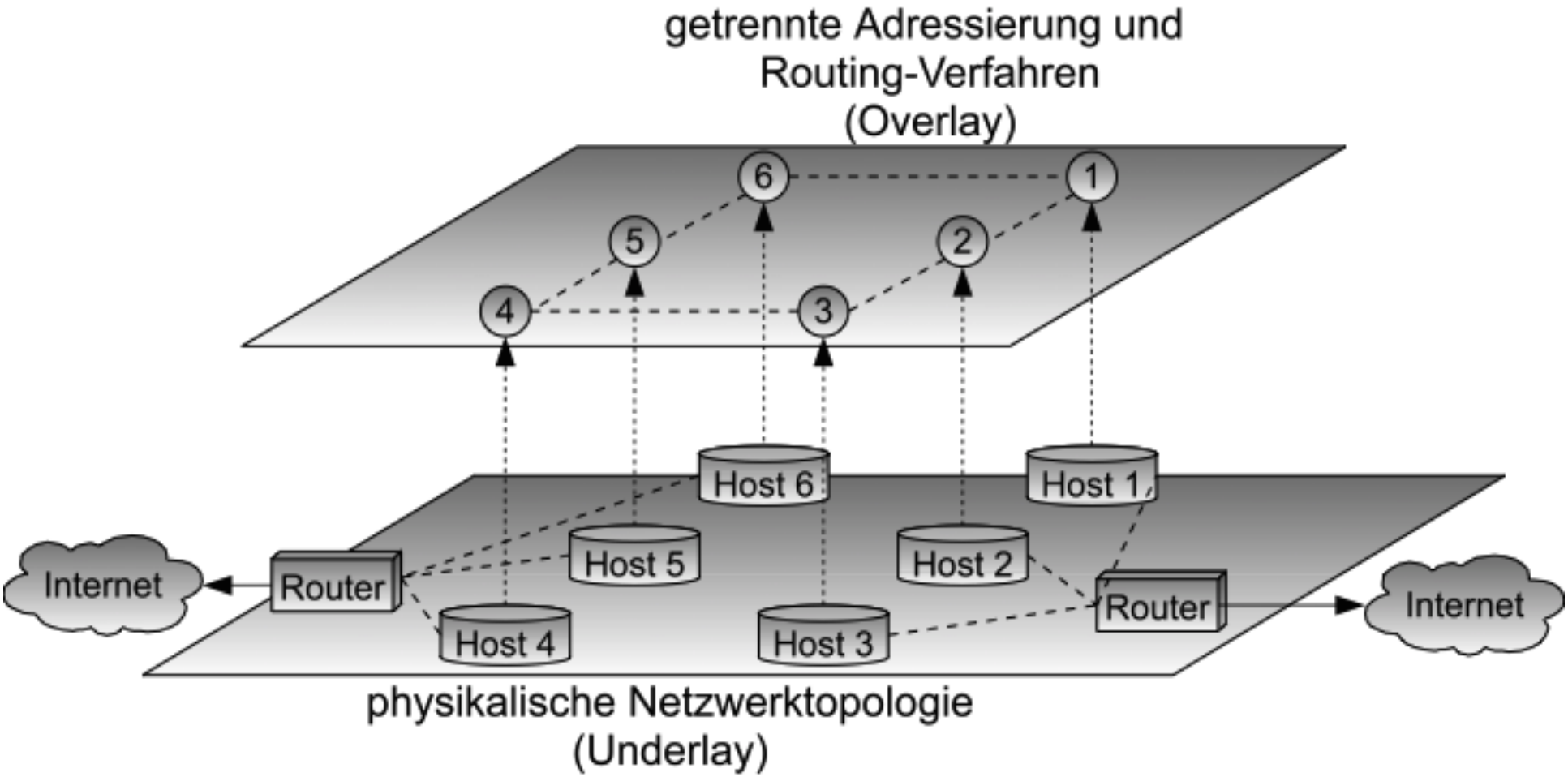


Client-Server Architektur



Peer-to-Peer Architektur

Peer-to-Peer Overlaynetzwerk



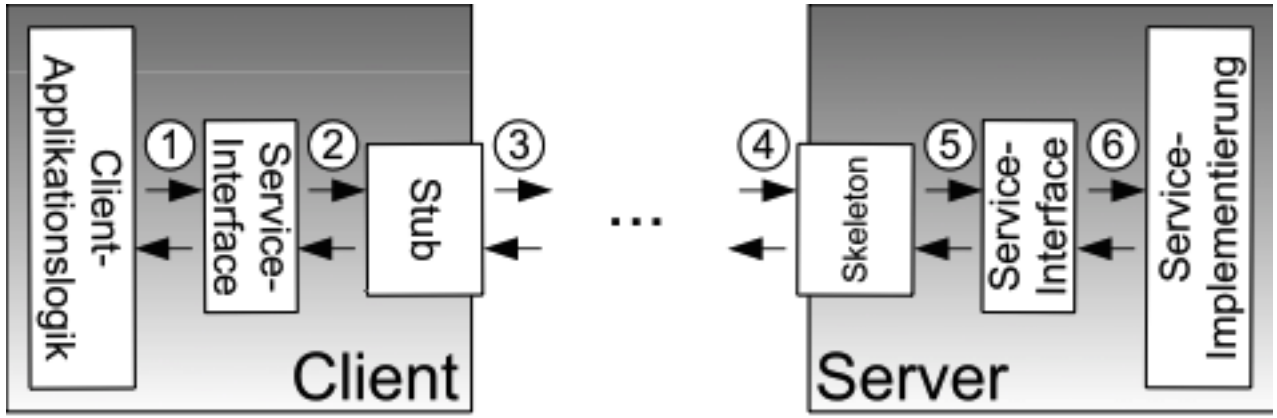
Uberlay

- *High performance small scale overlay network*
- Entwickelt am „Institut für Telematik“
- Basiert auf Netty und Google Protocol Buffers
- Routing:
  - Pfadvektor Protokoll
- Metrik:
  - Round Trip Time Protokoll

Remote Procedure Call

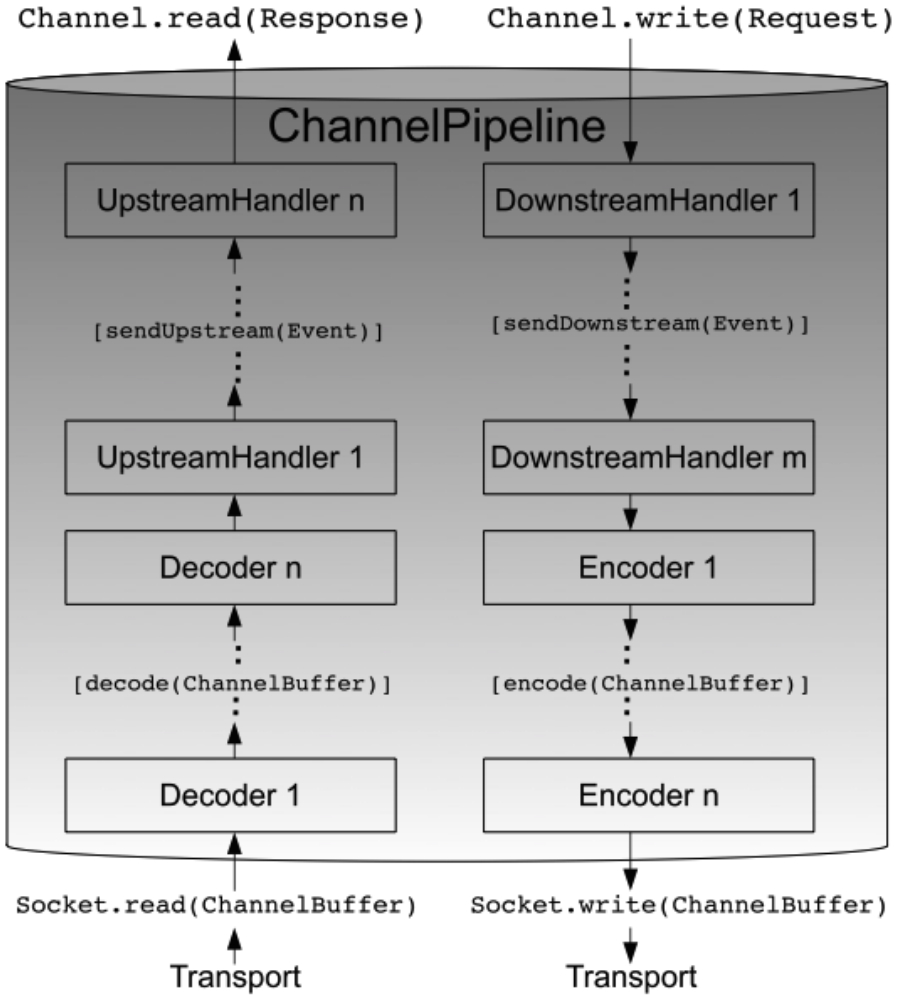
- Verfahren zur Abwicklung von Interprozesskommunikation
  - erlaubt Aufruf entfernter Prozeduren in verteilten Systemen
  - benötigt Marshalling / Unmarshalling

Ablauf:



Netty

- **Pipelines & Handler:** Handler in Pipeline sorgen mittels einem *Event-driven* Client-Server-Modell für Verarbeitung
- **Codecs:** Umwandlung der zu übermittelnden Objekte mittels Dekodern und Enkodern in übertragbare Nachrichtenpakete
- **Buffers:** Die Nachrichtenpakete werden in ChannelBuffers verpackt und versendet



Google Protocol Buffers

- Format zur Serialisierung von strukturierten Daten mittels einer IDL
  - Effiziente Serialisierung
  - Serialisierung mittels Beschreibung durch *proto*-Dateien
  - Generierung mittels eigenem Compiler: *protoc*

Beispiel

```
message BookOrderRequest{
    required Customer customer = 1;
    required Book book = 2;
    required bool deliverToCustomer = 3;
}

message BookOrderResponse{
    required uint64 orderNumber = 1;
}

message Customer{
    required uint32 customerNumber = 1;
    enum Gender{
        UNKNOWN = 0;
        MALE = 1;
        FEMALE = 2;
    }
    required Gender gender = 2 [default = UNKNOWN];
    required string name = 3;
    required string address = 4;
    optional uint32 phone = 5;
}

message Book{
    required string title = 1;
    repeated string authors = 2;
}
```

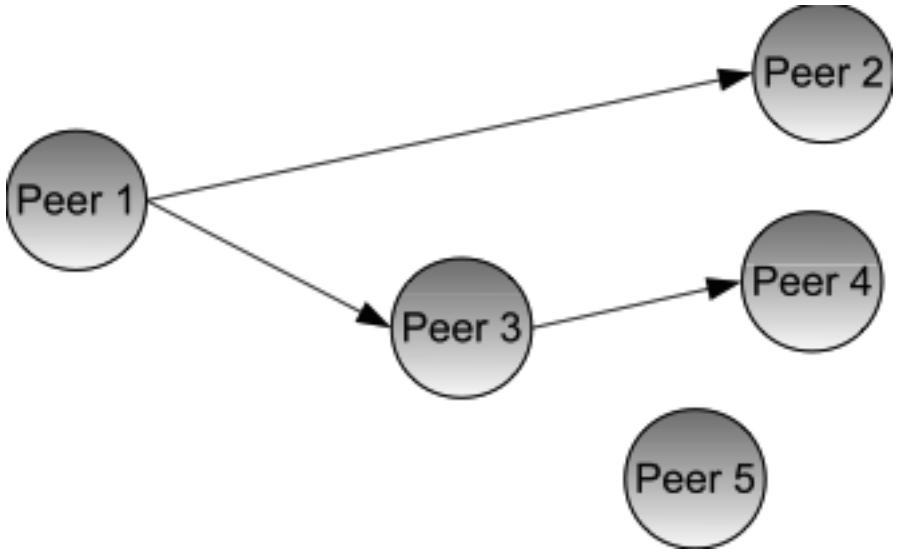
One-way

Unicast

- Message: 1
- Confirmation (optional): 1

Multicast

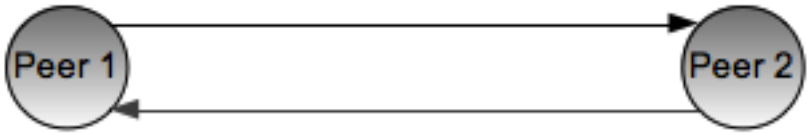
- Message: 1
- Confirmation (optional): 1..n



Request-Response

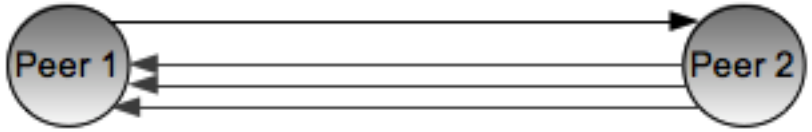
Single Request Single Response

- Request: 1
- Response: 1



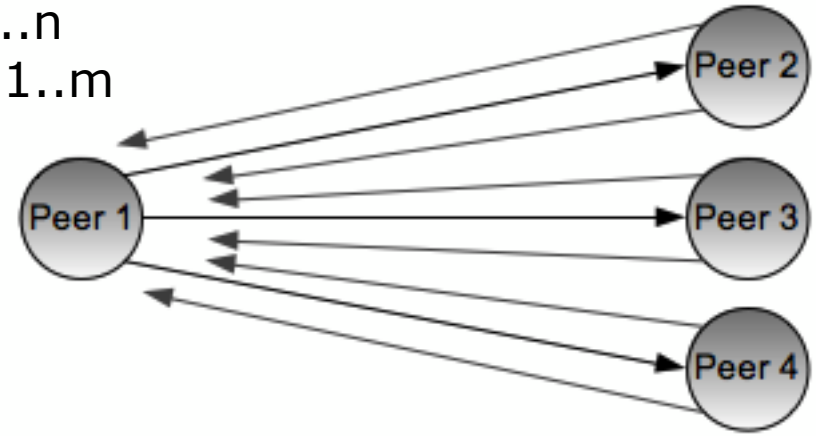
Single Request Multi Response

- Request: 1
- Response: 1..n

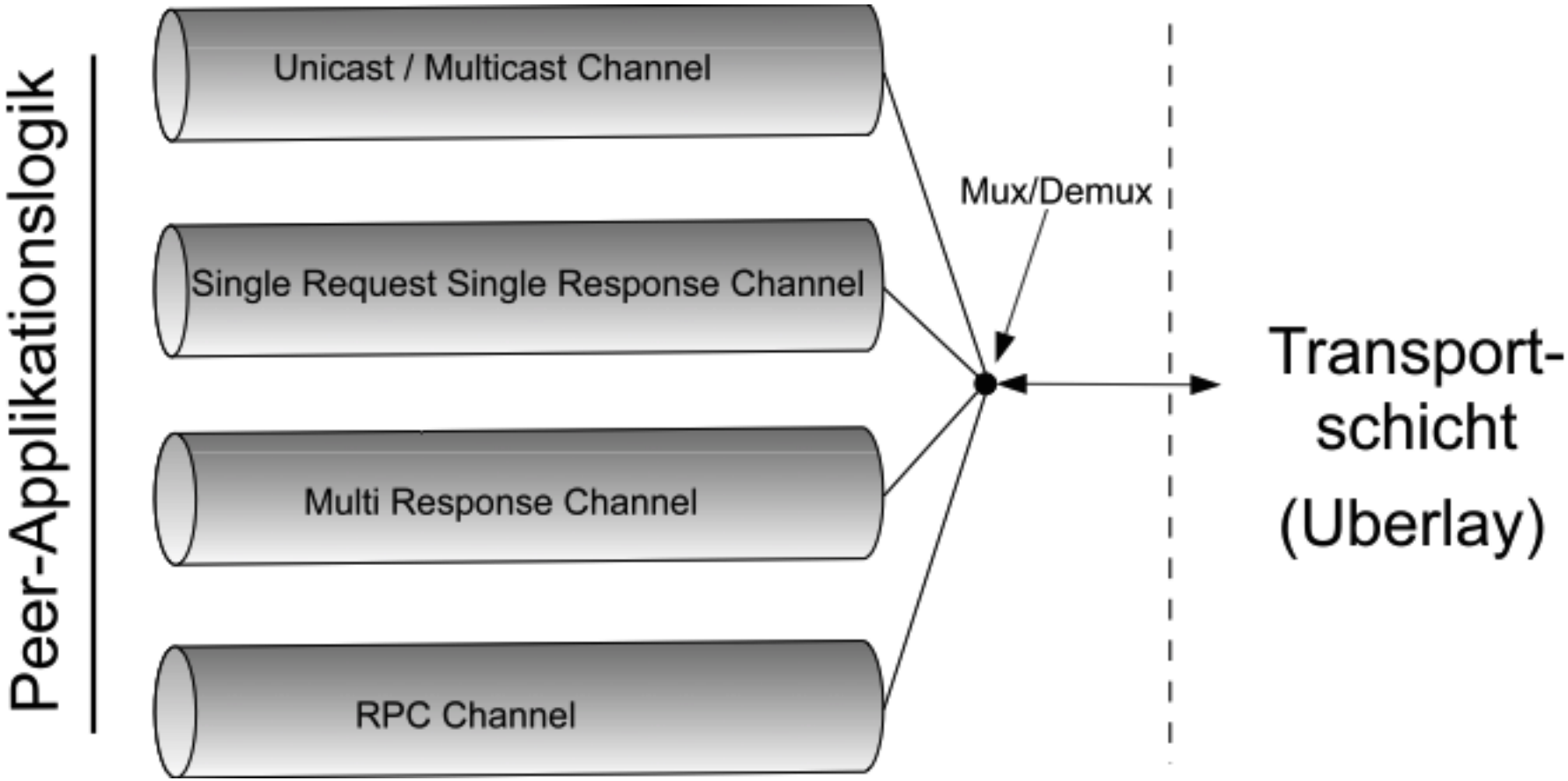


Multi Request Multi Response

- Request: 1..n
- Response: 1..m



Architektur



Definition

Protokoll:

ID	Typ	Reliable	Akt. Nachricht*	Anz. Nachrichten*	Payload

*: übertragen, nur wenn benötigt

Definition:
x := gesetzt
- := nicht gesetzt

One-way

Unreliable Unicast

Typ	Reliable
UNICAST	-

Unreliable Multicast

Typ	Reliable
MULTICAST	-

Reliable Unicast

Typ	Reliable
UNICAST	x

Reliable Multicast

Typ	Reliable
MULTICAST	x

Request-Response und RPC

Single Request Single Response

Typ	Reliable
SINGLE_RESPONSE_REQUEST	x

Typ	Reliable
SINGLE_RESPONSE	x

Multi Response

Typ	Reliable
MULTI_RESPONSE_REQUEST	x

Typ	Reliable	Akt. Nachricht	Anz. Nachrichten
MULTI_RESPONSE	x	x	x

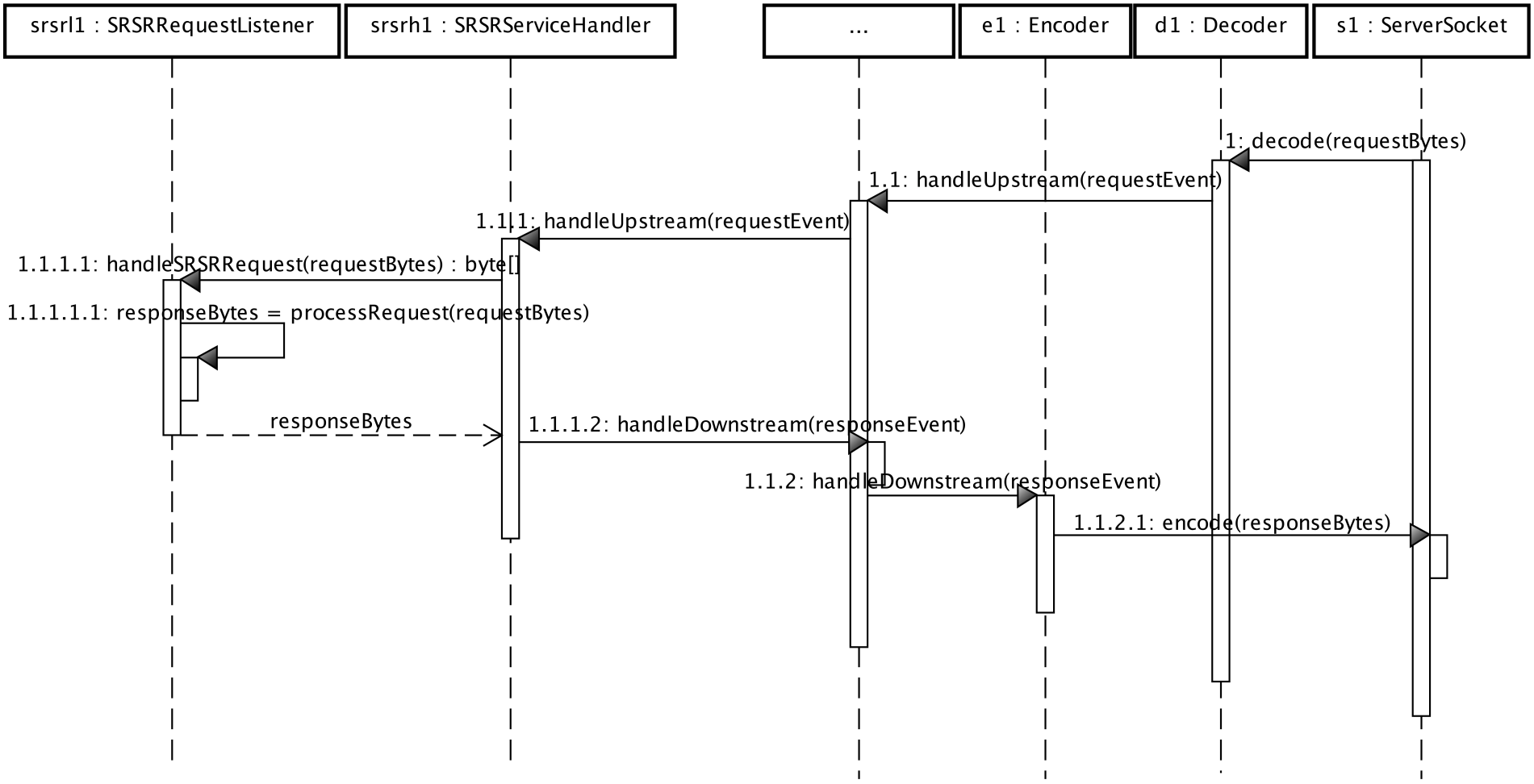
Remote Procedure Call

Typ	Reliable
RPC_REQUEST	-

Typ	Reliable
RPC_RESPONSE	-

Ubermep

Ablauf einer Single Request Single Response – Nachricht (Server-seitig)



Beispiele – Overlaynetzwerk erzeugen

- Server-seitig:

```
// Server erzeugen ...
Peer server = new PeerImpl(new UPAddress("urn:itm:1"), new InetSocketAddress("0.0.0.0", 8080));

// Server starten
server.start();

// Registrieren eines Listener am Peer
server.addRequestListener( new SingleRequestSingleResponseRequestListener() {
   public byte[] handleSingleRequestSingleResponseRequest(String senderUrn, byte[] requestPayload)
            throws UbermepExceptionEvent {
     // ... erzeugen der Response-Bytes ...
     return responseBytes;
   }
});
```

- Client-seitig:

```
// Client erzeugen ...
Peer client = new PeerImpl(new UPAddress("urn:itm:2"), new InetSocketAddress("0.0.0.0", 8081),
     new InetSocketAddress("0.0.0.0", 8080));

// Client starten
client.start();
```

Beispiele - Nachrichtenaustausch

```
ReliableRequest request = new SingleRequestSingleResponseRequest(address, requestBytes);
final ListenableFuture<Response> responseFuture = client.send(request);

// blockierender Aufruf
Response response = responseFuture.get();

// nicht-blockierender
responseFuture.addListener (new Runnable() {
    public void run() {
        // ... empfangene Response verarbeiten .., Bsp.:
        Response response = responseFuture.get();
    }
}, new ScheduledThreadPoolExecutor(1));

// zusätzlich für Multi-Response-Nachrichten:
responseFuture.addListener(new ProgressListenerRunnable() {
    public void singleResponseReceived(String senderUrn, byte[] payload, int current,
            int total) {
        // ... einzeln empfangene Response verarbeiten ...
    }
    public void run() {...}
}, new ScheduledThreadPoolExecutor(2));
```

Beispiele - RPC

- Generieren eines „Protobuf-RPC-Service“ mittels Google Protocol Buffer-IDL und -Compiler
- Implementieren des („Protobuf“) `Interface/BlockingInterface` und des („Ubermep“) `RpcService/RpcBlockingService`
- Registrieren des implementierten Services mittels `registerService(...)` am (Server-) Peer
- Aufruf über einen `UbermepRpcChannel` durch `getRpcChannel(address)` am (Client-) Peer
- Erzeugen eines Stubs mithilfe des `UbermepRpcChannel` (Client-seitig)
- Ausführen mittels direktem Aufruf bzw. `MEPRpcCallback`-Implementierung über Stub

Evaluationsszenarien

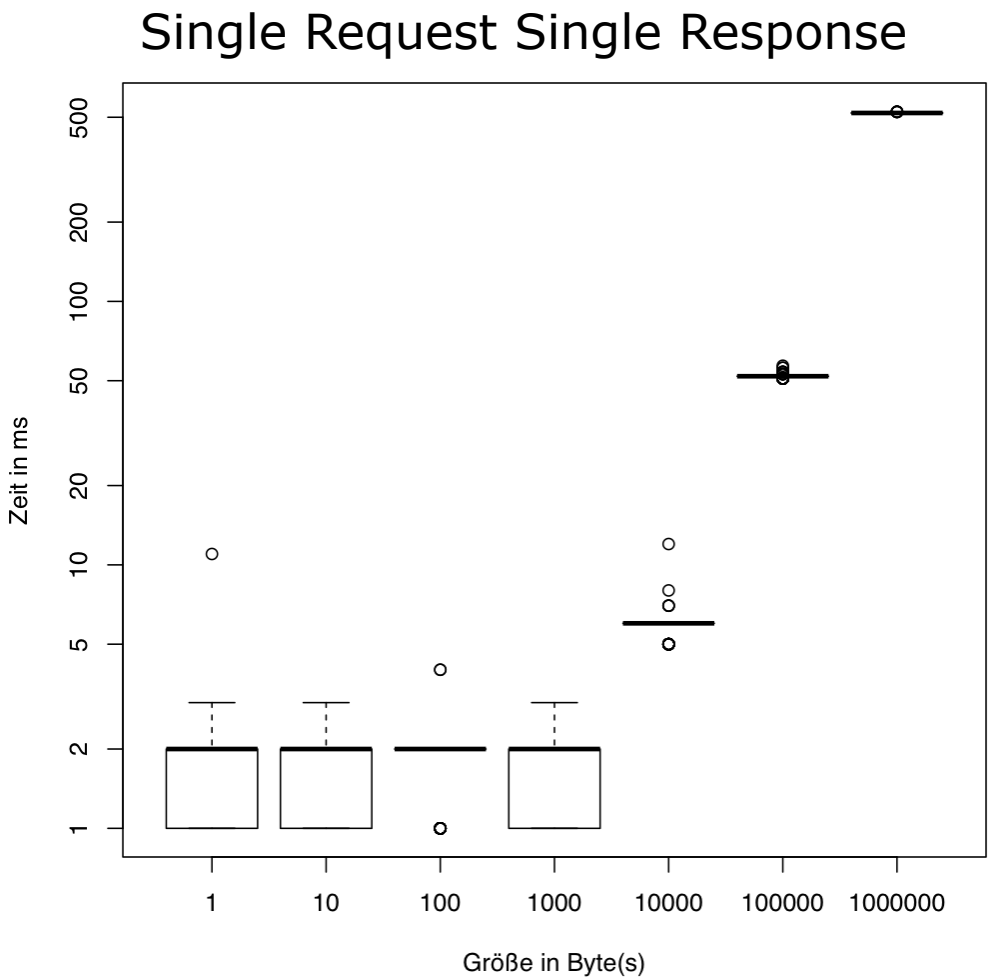
Topologien:

Übertragungsdauer in Abhängigkeit der Payload-Größe:

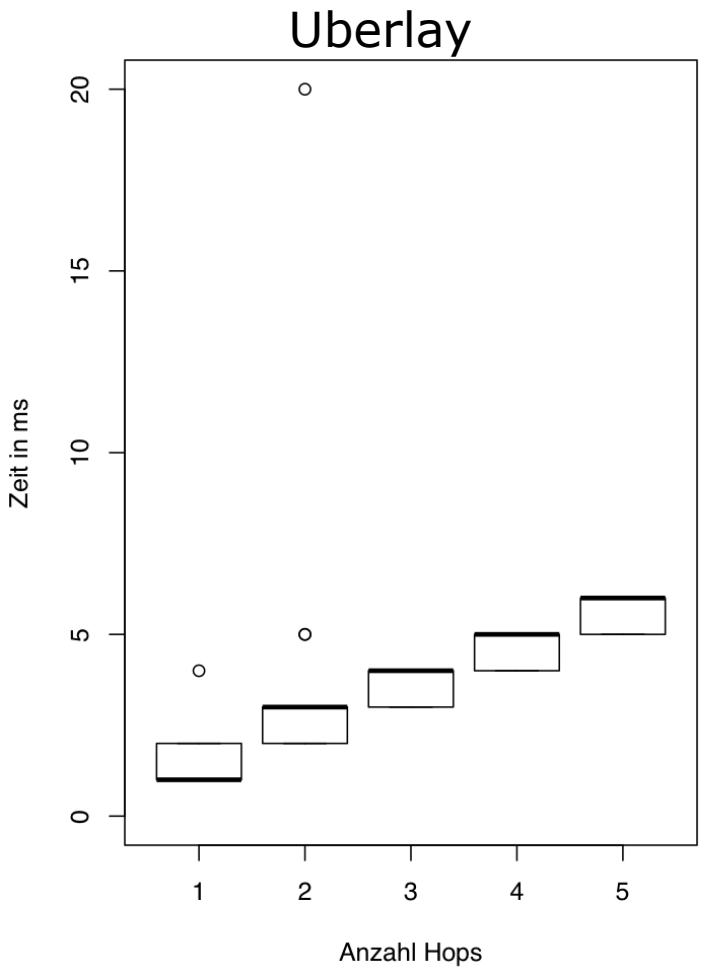
Übertragungsdauer in Abhängigkeit der Hop-Anzahl:

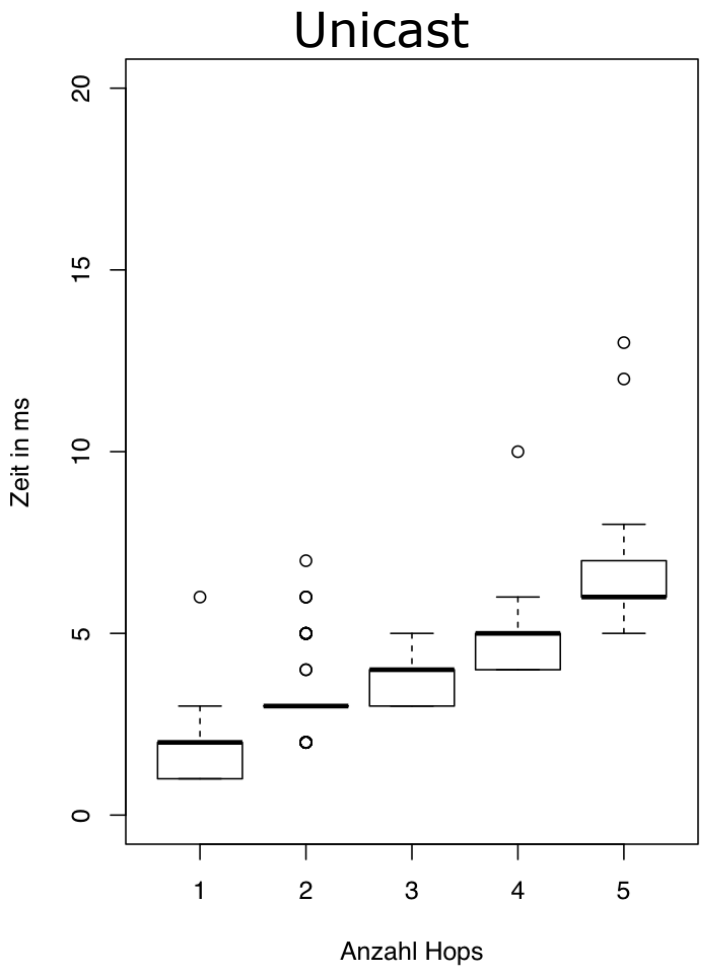
- Verwendung von:
  - Loopback-Interface
  - Emulator für Übertragungsrate von 16 MBit/s

Ergebnisse: Single Request Single Response



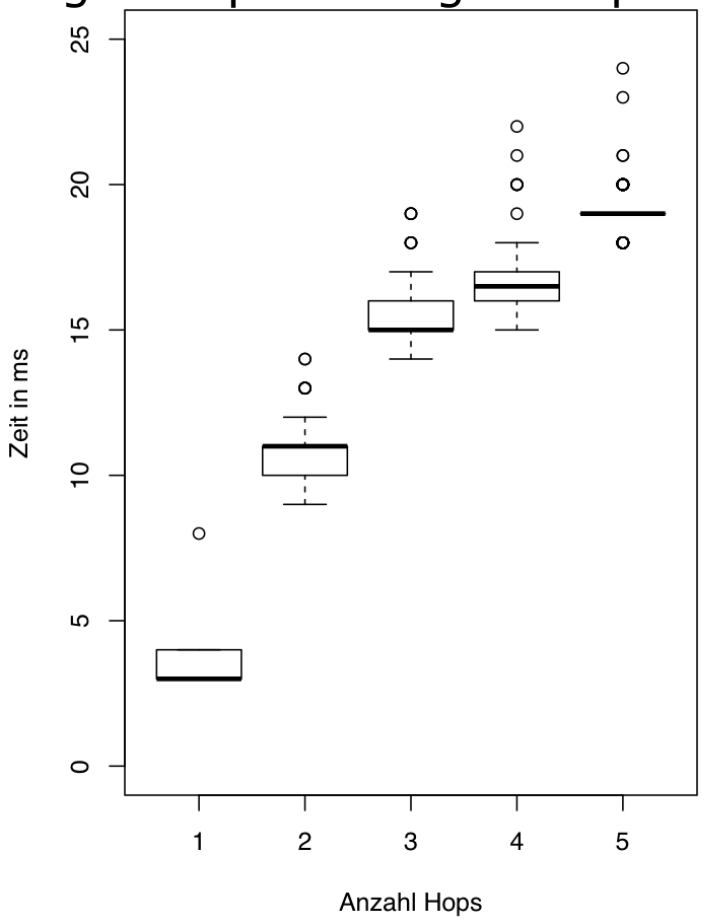
Ergebnisse: Uberlay und Unicast



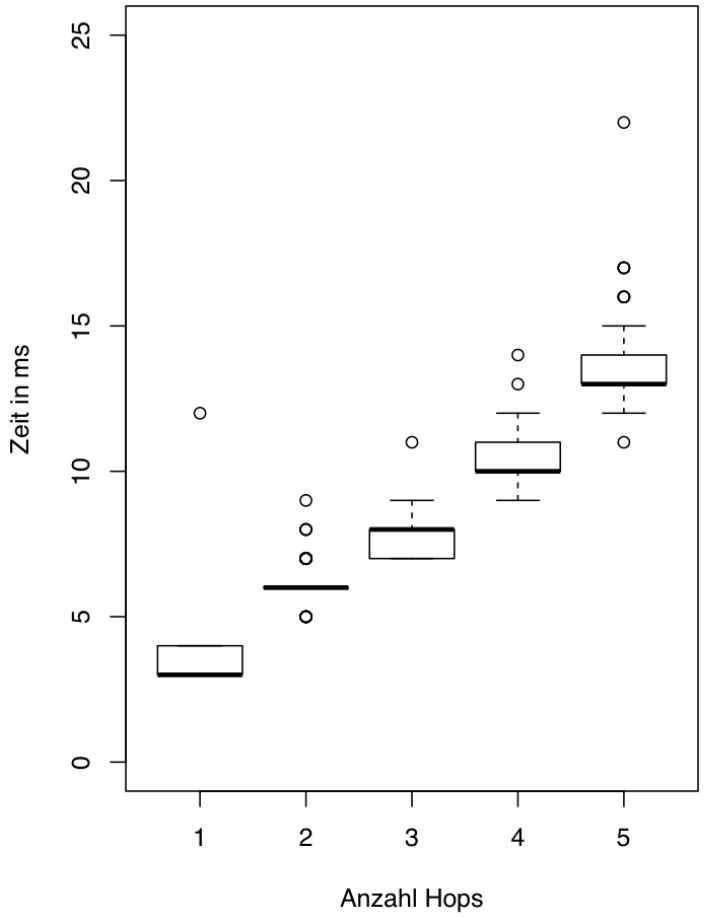


Ergebnisse: Single Request Single Response und RPC

Single Request Single Response:



Remote Procedure Call:



Zusammenfassung & Ausblick

- Ergebnis dieser Arbeit:
  - Middleware-Lösung zum Nachrichtenaustausch in einem Peer-to-Peer-basierten Overlaynetzwerk
  - Kommunikation in Testbed-Runtime sowie weiteren Anwendungsbereichen gegeben
- Durch Evaluation gezeigt:
  - Gute Performanz, (impliziert Funktionalität)
- Zukunft:
  - Implementierung des *Publish-Subscribe Pattern*

Vielen Dank für Ihre Aufmerksamkeit!

Für Interessierte:

Implementierung unter: *https://github.com/nrohwedder/ubermep*