Praktikum
Mobile und Verteilte Systeme

Prof. Dr. Claudia Linnhoff-Popien
Moritz Kessel, Philipp Marcus

Sommersemester 2012
Remote Procedure Calls and Interprocess Communication

• Today:
  – Characteristics of Distributed Systems
  – Socket Communication (Focus on Java)
  – Middleware
  – Request/Reply Communication and Remote Procedure Calls
  – Apache Thrift (RPC-Framework)

• Next Week:
  – Distributed Objects
  – Remote Method Invocation
  – Java RMI
Examples of Distributed Systems

- The Internet is the largest distributed system, consisting of intranets, subnetworks, backbones, ISPs...
- Connected computers interact by passing messages
- The internet protocols are a common means of communication
- Allows the interaction of two separate processes
- Services are provided with an open end for:
  - Multimedia (Radio, TV, phone calls...)
  - File transmission
  - And many more...

Source: [http://navigators.com/sessphys.html](http://navigators.com/sessphys.html)
Definition of Distributed Systems

- **Distributed system**: set of processes that communicate only by message passing – not using a shared memory. (Couloris et al.)
Advantages of Distributed Systems

- **Resource sharing**: Allows the shared usage of resources like printers, files, web pages, database records...
- **Openness**: Most systems are designed as open systems and Hard- and Software of different types can be combined.
- **Concurrency**: Processes can run concurrently on different machines and are able to communicate.
- **Scalability**: Installation of new computers increases capacity – network bandwidth is a limiting factor.
- **Failure handling**: Failures are typically partial: one component fails and the others continue working.
Communication Channels (1)

- **message passing**: enables a sending process to transmit a single message to a receiving process
- Messages are transmitted using a communication channel as a sequence of bytes

![Diagram of message passing](image)

- `send(q,m)` transmits a message `m` to a receiving process `q`
- `receive(m)` receives a message `m`
- Synchronous transmission: Blocking `send` and `receive`
- Asynchronous transmission: Non-blocking send and blocking or non-blocking receive
Communication Channels (2)

- communication channels are **not perfect**:  
  - it takes (an arbitrary) time to transmit the message  
  - it takes (an arbitrary) time to process incoming messages  
  - clocks of communicating computers may drift arbitrarily much

- synchronous distributed system:  
  - upper bound for transmission time  
  - upper bound for processing delay  
  - upper bound for clock drift

- asynchronous distributed system:  
  - at least one size is not limited by an upper bound

**Question**: Is the internet a synchronous or an asynchronous medium?
Reliability of one-to-one communication (1)

- Reliability of one-to-one communication is defined in terms of **integrity** and **validity**
- **integrity:**
  - the message received is identical to the one sent
  - no messages are delivered twice
- **validity:** any message in the outgoing message buffer is eventually delivered to the incoming message buffer
Reliability of one-to-one communication (2)

- Threats to integrity:
  - injection of spurious messages by malicious users
  - malicious users that tamper with messages or replay old ones

- Threats to validity:
  - communication channels might drop messages (malfunction or congestion of segments)
Using the Internet Protocols

- Sockets are used by UDP and TCP as endpoints between communicating processes

- Sockets are:
  - associated to an IP-Address, a port and a single process
  - used for sending and receiving data
  - bound to UDP or TCP (Transport Layer)

- Many clients can connect to a socket – connections are defined by (IP-Client, Port-Client, IP-Server, Port-Server)
TCP Stream Communication

- Data transmission is represented as a stream and read or written for send and receives
- Following network characteristics are hidden:
  - **Message sizes**: Implementation decides how much data is collected before an IP-packet is transmitted
  - **Lost messages**: ACK packages are sent transparently and confirm packet reception. Retransmission is issued if no ACK arrives.
  - **Flow control**: Matching of process speeds. Too fast writers are blocked.
  - **Message duplication and ordering**: Message Ids allow reordering of packets and detection of duplicates.
  - **Message destinations**: After a `connect` from the client and an `accept` from the server, communication can take place without need for internet address or port numbers
- Initiation: A client connects to a known port the server is listening on. On accept a new stream socket is connected and the server continuous listening for connects of other clients.
Communication in Java: Sockets (1)

Client

```java
public class Connect {
    public static void main(String[] argv) {
        String server_name = "localhost";
        try {
            //create socket
            Socket sock = new Socket(server_name, 9999);
            System.out.println("connected");
            //close socket
            sock.close();
        } catch (java.io.Exception e) {
            System.out.println(e);
            return;
        }
    }
}
```

Server

```java
public class Connect {
    public static void main(String[] argv) {
        ServerSocket sock;
        Socket clientSock;
        try {
            sock = new ServerSocket(9999);
            while (clientSock = sock.accept() != null) {
                //Serve Client
                process(clientSock);
            }
        } catch (java.io.Exception e) {
            System.out.println(e);
        }
    }
    static void process(Socket s) throws IOException {
        System.out.println("Client verbunden: " + s.getInetAddress());
        //... Code for client communication...
        s.close();
    }
}
```

Iteratively wait for connecting clients
Communication in Java: Sockets (2)

- **Reading text from Sockets (single line):**
  ```java
  BufferedReader is = new BufferedReader(new InputStreamReader(sock.getInputStream()));
  //read a string from the peer
  String read = is.readLine();
  ```

  And for multiple lines:
  ```java
  String received;
  while ((received = is.readLine()) != null) {
    //process received string
    //wait for new string...
  }
  ```

- **Writing text to sockets:**
  ```java
  PrintWriter os = new PrintWriter(sock.getOutputStream(), true);
  //transmit a string to the peer
  os.print("test\r\n");
  os.flush();
  ```

- *flush()* is important for sending out buffered data
- Recommended for strings: Terminate single commands with carriage return and newline symbol: \r\n
- Binary data could be processed by DataInputStream and DataOutputStream
Communication in Java: Sockets (3)

- Handling of multiple clients is implemented using **threads**
- Create a Handler-Class as a subclass of class Thread or Runnable:

```java
class Handler extends Thread {
    Socket sock;
    Handler(Socket s) {
        sock = s;
    }

    public void run() {
        System.out.println("Handler started...");
        //Here the server code with readLine etc.
    }
}
```

```java
void runServer() {
    while (true) {
        try {
            Socket clntSock = sock.accept();
            new Handler(clntSock).start();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```
The Client/Server Paradigm

- Client/Server architecture is a paradigm often used in distributed systems
- Consists of a set of servers with services and a set of interconnecting clients

- Thin-Clients: Application processes and data management run on the server (for data centric browsing or querying)
- Fat-Clients: The server only implements the data management (applications with complex computation of data – i.e. visualization)
- Three-Layer C/S systems are often used for complex and comprehensive systems (i.e. client <-> webserver <-> sql server)
- Clients often access services using remote procedure calls
Request/Reply Communication (1)

- Request-Reply-Communication with three basic operations

- Typically implemented as synchronous communication
  - `doOperation` marshalls the data and issues a `send`
  - `getRequest` acquires a client request via the server port
  - `sendReply` sends the reply message to the client at its port and IP
Request/Reply Communication (2)

- Request-Reply-Messages that focus on executing a specific method on the peer are called **Remote Procedure Call**
- Hide communication by making remote calls look like lokal ones

<table>
<thead>
<tr>
<th>messageType</th>
<th>int (0=Request, 1=Reply)</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestID</td>
<td>int</td>
</tr>
<tr>
<td>methodID</td>
<td>int or method</td>
</tr>
<tr>
<td>arguments</td>
<td>array of bytes</td>
</tr>
</tbody>
</table>

- **messageType**: Indicates if messages is a request or reply
- **requestID**: Allows the client to check if a response corresponds to its latest call
- **methodID**: Identifier for the method to be invoked. Methods of an interface can be numbered (1,2,3,...)
- **arguments**: Arguments of the method
Data format needs to be defined **Definition (Marshalling):** Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message (Couloris et al.)

**Definition (Unmarshalling):** Unmarshalling is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination.

**Definition (External Data Representation):** Flattening of complex data types and representing primitive datatypes in an agreed format (i.e. Little Endian, Big Endian, UTF, ASCII...).
Remote Procedure Calls (1)

- **A stub procedures** on the provide this interface for local calls and **internaly forward them to a server skeleton**
- Typically generated automatically based on the interface definition
- The stub works like a proxy:
  - Receive a local call, marshall the call and the parameters, send a request to the server process
- The server skeleton has the following tasks:
  - Determine the correct service procedure, unmarshall the arguments, locally call the service procedure, marshall the results, send to client
Remote Procedure Calls (2)
Remote Procedure Calls (3)

- The application itself should not need to concern about communication details – it is not visible to the application that a procedure is remote
- Interface of remote methods is provided by a specification in an Interface Definition Language (IDL) and defines the service interface
- Apache Thrift, Google Protocol Buffers, Sun XDR, OMG IDL
- Time server example for Apache Thrift:
  ```
  # time.thrift
  namespace java tserver.gen
  typedef i64 Timestamp
  service TimeServer {
    Timestamp time()
  }
  ```
- Many more IDLs are available; targeting at platform independency
- Client and server need to implement these interfaces...

**Question:** Is it possible to pass references as arguments?
Request/Reply Communication (3)

- **RPC Failure semantics:**
  1. **Client cannot locate the server**
     - Solve with exceptions; destroys transparency
  2. **Lost Request Message**
     - Timer with exceptions; not relevant for TCP
  3. **Server crashes:**
     - **normal case:**
     - **crash after execution:**
     - **crash before execution:**

  ![Diagram of request/reply communication](image)

  4. **Lost Reply Message**
  5. **Client crashes**

*Should the client always re-send the request after timeout?*
Request/Reply Communication (4)

- The client can react to server crashes in different ways, resulting in different **invocation semantics**:

<table>
<thead>
<tr>
<th>Semantic of procedure invocations</th>
<th>The request is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>At least once</td>
<td>sent until a reply is received</td>
</tr>
<tr>
<td>At most once</td>
<td>not sent again and an error is reported immediately</td>
</tr>
<tr>
<td>Exactly once</td>
<td>the desired semantics but not possible</td>
</tr>
<tr>
<td>Maybe</td>
<td>Not sent again. The client gets no response about the execution of the request</td>
</tr>
</tbody>
</table>

- Example: Imagine a printing server that crashed, rebooted and informed the clients about the reboot...
Automatic Stub Generation and De-/Serialization

RPC Frameworks provide the essential middleware components, here some modern examples:

• Apache Thrift
  – Offers a framework for serialization/deserialization and a stack for RPC calls
  – Originally developed by Facebook and contributed to Apache (Apache 2.0 license)
  – Currently 12 languages are supported (Java, C++, C#, Ruby...)

• Google Protocol Buffers
  – Format for data serialization
  – Developed to solve problems with versioning
  – Can create simple stubs but does not provide RPC implementations for client and server
  – More robust and consistent than Thrift but has less features

• Apache Etch, XML-RPC...
Thrift Overview (1)

- Thrift allows the definition of service interfaces, a group of procedures that are provided by a server.
- A IDL is provided to define complex data structures and service interfaces.
- Client can even asynchronously invoke operations.
- Client and Server consist of three main layers: written Code, generated code and the Apache Thrift libraries.
- The written code allows to choose which marshalling mechanisms (TProtocol) and which I/O mechanisms are used (TTransport).

Thrift Overview (2)

- Transport defines „how is transmitted“ in terms of I/O operations
- Application code should run against TCP stream sockets, raw data in memory or files on disk
- `TTransport` is a interface for I/O with methods like `open`, `close`, `isOpen`, `read`, `write` and `flush`
- Thrift transports (implementations of `TTransport` interface)
  - `TSocket`: Implements the above presented `java.net.Socket` class using `InputStream` and `OutputStream`
  - `TNonblockingSocket`: Implements `java.nio.channels.SocketChannel` for non-blocking I/O Operations (using OS socket implementations)
  - `TFrameTransport`: Transmits data with frame size headers for chunking optimization or nonblocking operation – opposite to stream communication
  - `TFileTransport`: Abstraction of an on-disk-file to a data stream.
  - `TMemoryTransport`: Allows directly reading from the heap or stack memory owned by the process
Thrift Overview (3)

• The Thrift protocols define „what is transmitted?“, i.e. how datatypes are encoded and decoded (Marshalling) for transmission over a Transport and how messages are constructed

• The application itself is separated from this layer, i.e. it is immaterial to the application code if XML, a textual or a binary format is used

• The interface TProtocol provides methods for composing and decomposing messages like writeMessageBegin, writeMessageEnd, writeStructBegin, writeStructEnd, writeBool, writeMapBegin... and corresponding read methods

• Thrift protocols (implementations of TProtocol)
  – TBinaryProtocol: Numeric values are encoded binary instead of text
  – TCompactProtocol: Efficient and dense encoding of data
  – TDebugProtocol: A human readable text format is used
  – And many more...

• A possible sequence could be: writeMessageBegin, writeStructBegin, writeFieldBegin, writeFieldEnd, writeStructEnd, writeMessageEnd
Thrift Overview (4)

- The real service procedures are encapsulated in a `TProcessor` object in the generated code.
- This code is passed to an instance of `Tserver`, existing code to easily set up servers.
- These servers do the connection handling, thread management.
- Example servers, deployed at Facebook:
  - `TSimpleServer`: Single threaded server
  - `TThreadPoolServer`: one thread per connection server
  - `TThreadedServer`: one thread per connection with a thread-pool
- For initiation the server also needs a `TSocket`, a `TTransport` and a `TProtocol`.

**Inheritants of TServer(AbstractServerArgs) as parameter for server constructors**
The Thrift Interface Definition Language (1)

- The Thrift IDL provides several primitive datatypes to construct complex structured datatypes or to be used in interface specifications:

<table>
<thead>
<tr>
<th>Thrift type</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Boolean, one byte</td>
</tr>
<tr>
<td>byte</td>
<td>Signed byte</td>
</tr>
<tr>
<td>i16</td>
<td>Signed 16-bit integer</td>
</tr>
<tr>
<td>i32</td>
<td>Signed 32-bit integer</td>
</tr>
<tr>
<td>string</td>
<td>String</td>
</tr>
<tr>
<td>map&lt;t1, t2&gt;</td>
<td>Map from one type to another</td>
</tr>
<tr>
<td>list&lt;t1&gt;</td>
<td>Ordered list of one type</td>
</tr>
<tr>
<td>set&lt;t1&gt;</td>
<td>Set of unique elements of one type</td>
</tr>
<tr>
<td>...and some more (double, binary, i64)</td>
<td></td>
</tr>
</tbody>
</table>

- Typedef for pretty names, i.e.: `typedef i32 MyInteger`
The Thrift Interface Definition Language (2)

- **Enumerations** are supported with optional values (default starting at 1):
  ```thrift
enum Color {
    GREEN = 1,
    RED = 2,
    BLUE = 3
  }
```

- **Structs** are used to define complex datatypes consisting of primitives, enums and structs:
  ```thrift
  struct Person {
    1: required Color favoritColor, type integer identifier
    2: string name, symbolic name
    3: i16 age,
    4: i16 friends = 0
  }
```

- **Exceptions** are special structs:
  ```thrift
  exception PersonExistingException {
    1: string errorMessage,
    2: i32 errorId
  }
```
The Thrift Interface Definition Language (3)

- A Thrift specification defines a service:
  ```thrift
  service AddressBookService extends shared.SharedService {
    list<Person> getPersons(),
    void add(1:Person person) throws (1:PersonExistingException pee),
    oneway void zip()
  }
  ```

- Namespaces are supported:
  ```thrift
  namespace java ifi.lmu.mobile.thrift.gen
  ```

- More details can be found at [http://thrift.apache.org/docs/idl/](http://thrift.apache.org/docs/idl/)
Thrift Code Generation

• The thrift compiler generates Stubs and Skeletons, Java classes and all needed files
• It is invoked by the command:
  
  thrift --gen java addressbook.thrift

• Individual files for structs and exceptions are created – enums are not mapped to java enums
• In our example, the following code structure is generated:

```
PersonExistingException.java
Person.java
AdressbookService.java
```
Writing a Thrift Server

• The framework provides servers that need a processor (generated from the IDL), a socket, a protocol and a transport

• Code for setting up the connection and starting the server:

```java
final TServerSocket socket = new TServerSocket(PORT);
final AdressbookService.Processor processor = new AdressbookService.Processor(
    new Handler());
//...for a server using a thread pool
final TServer server = new TThreadPoolServer(new TThreadPoolServer.Args(socket).processor(processor));
//...a simple server supporting only one client simultaneously
final TServer server = new TSimpleServer(new TServer.Args(socket).processor(processor));
//...a nonblocking server internally using java nio
final TServer server = new TNonblockingServer(new TNonblockingServer.Args(socket).processor(processor));
server.serve();
```

• Code that implements the interface and the service procedures:

```java
static class Handler implements AdressbookService.Iface {
    @Override
    public List<Person> getPersons() {
        //... to be implemented
    }
    //...other procedures need to be implemented too
}
```
Writing a Thrift Client

- Code for setting up the transport and the protocol:

```java
final TSocket socket = new TSocket(HOST, PORT);
socket.setTimeout(SOCKET_TIMEOUT);
final TTransport transport = new TFramedTransport(socket);
final TProtocol protocol = new TCompactProtocol(transport);
final AddressbookService.Client client = new AddressbookService.Client(protocol);

Choose the same TTransport and TProtocol as the server

Automatically generated for sync. communication

- Opening the transport:

```java````
```java
transport.open();
````

- Code for using the service:

```java````
```java
List<String> persons = client.getPersons();
System.out.println("Received " + persons.size() + " persons.");

client.add(new Person("BLUE","Mr. Black", 31, 153));
List<String> persons = client.getPersons();
System.out.println("Received " + persons.size() + " persons.");
````

```java````
```java
transport.close();
````
Writing Asynchronous Clients (1)

- Up to now: Synchronous Calls, i.e. the calling thread is blocked until the server returned the result:

  - Thrift also supports asynchronous calls in clients, i.e., a call of a method immediately returns though the result has not yet been received.

  - A extra handler is registered when the async. call is made
  - The handler processes the returned result in an extra thread (using Java NIO)
  - The original thread continuous running
Writing Asynchronous Clients (2)

- **Step 1:** Define a Handler:

```java
class GetPersonsMethodCallback implements AsyncMethodCallback<AddressbookService.AsyncClient.getPersons_call> {
    public void onComplete(AddressbookService.AsyncClient.getPersons_call returnValue) {
        try {
            System.out.println(returnValue.getResult());
        } catch (TException e) {
            //...
        }
    }
    public void onError(Exception e) {
        //implementation
    }
}
```

- Handler implements AsyncMethodCallback
- Two methods: `onComplete` and `onError` as callback methods

- **Step 2:** Instantiate a client:

```java
AddressbookService.AsyncClient client = new AddressbookService.AsyncClient(
    new TBinaryProtocol.Factory(),
    new TAsyncClientManager(),
    new TNonblockingSocket("localhost", 7911));
```

- TNonblockingSocket and same TProtocol as the server
- Only non-blocking servers are supported by the current implementation
Writing Asynchronous Clients (3)

- **Step 3:** Make a call:
  ```java
  client.getPersons(new GetPersonsMethodCallback());
  ```
  - A new client instance is needed for every asynchronous call (can only handle one request at once)
  - Async. Clients only work with Non-Blocking servers by now (i.e. TNonblockingServer or THsHaServer)
  - Requests are handled in a separate „Selector“-Thread (compare to Java NIO = non-blocking I/O)
  - Further asynchronous method invocations need a fresh instance of AddressbookService. AsyncClient
References & Literature

- Couloris et al.: "Distributed Systems - Concepts and Design" 3rd Edition
- Andrew S. Tanenbaum et al.: "Distributed Systems - Principles and Paradigms"
- Ian F. Darwin: "Java Cookbook" 2nd Edition
- Mark Slee et al., „Thrift: Scalable Cross-Language Services Implementation“
- http://floatingsun.net/articles/thrift-vs-protocol-buffers/
- http://thrift.apache.org/docs/
Practical Course

• Prepare Eclipse for Apache Thrift
  – Compiling, Installing...

• Experiment with Apache Thrift versioning techniques
  – Outdated client protocol
  – Outdated server protocol...

• Implement a RPC-based chat system
  – Define an interface definition
  – Implement the server and client
  – Experiment with asynchronous procedure calls