GEDISTRIBUEERDE TOEPASSINGEN

Veerle Ongenae

Inhoud cursus

2

- Webservices
  - Gedistribueerde applicaties
  - SOAP
  - WCF
- Netwerkprogrammatie Java
- JMS
- Node.js en Websockets

Quotatie

3

- Theorie (50%)
  - Mondeling examen in januari
- Labo (50%)
  - 2 testen (3/11 en 15/12)
  - Aanwezig – Voorbereiden
  - Labopunten behouden in 2de zit!

Cursusmateriaal

4

- Leerpad (minerva)
  - Overzicht webpagina's met informatie
- Slides (notitiepagina's) en voorbeeldprogramma's
  - Minerva - documenten

Verschuiving lessen

5

- Theorie ipv labo
  - Di 22/09 13u30 – 15u00
- Geen les
  - Di 11/11
- Extra labo
  - Di 15/12

JAVA NIO

Veerle Ongenae

Java NIO

2

- Java New IO
- Alternatief
  - Java IO
  - Java Networking

Industrieel Ingenieur Informatica, UGent

Java NIO (New IO) is an alternative IO API for Java (from Java 1.4), meaning alternative to the standard Java IO and Java Networking API's. Java NIO offers a different way of working with IO than the standard IO API's.

Java NIO: Channels and Buffers

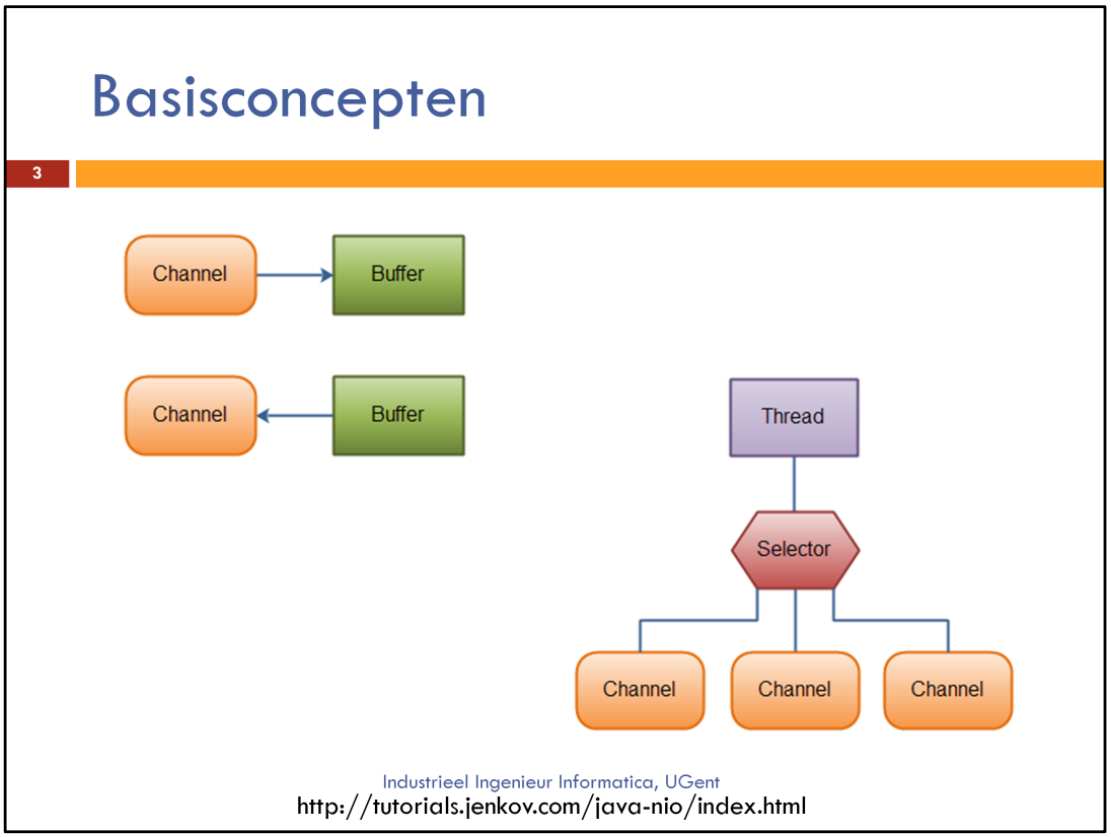
In the standard IO API you work with byte streams and character streams. In NIO you work with channels and buffers. Data is always read from a channel into a buffer, or written from a buffer to a channel.

Java NIO: Non-blocking IO

Java NIO enables you to do non-blocking IO. For instance, a thread can ask a channel to read data into a buffer. While the channel reads data into the buffer, the thread can do something else. Once data is read into the buffer, the thread can then continue processing it. The same is true for writing data to channels.

Java NIO: Selectors

Java NIO contains the concept of "selectors". A selector is an object that can monitor multiple channels for events (like: connection opened, data arrived etc.). Thus, a single thread can monitor multiple channels for data.



Java NIO consist of the following core components:

Channels
Buffers
Selectors

Java NIO has more classes and components than these, but the Channel, Buffer and Selector forms the core of the API. The rest of the components, like Pipe and FileLock are merely utility classes to be used in conjunction with the three core components.

Channels and Buffers

Typically, all IO in NIO starts with a Channel. A Channel is a bit like a stream. From the Channel data can be read into a Buffer. Data can also be written from a Buffer into a Channel. There are several Channel and Buffer types. Here is a list of the primary Channel implementations in Java NIO:

FileChannel
DatagramChannel
SocketChannel
ServerSocketChannel

As you can see, these channels cover UDP + TCP network IO, and file IO.

Here is a list of the core Buffer implementations in Java NIO:

ByteBuffer
CharBuffer

DoubleBuffer
FloatBuffer
IntBuffer
LongBuffer
ShortBuffer

These Buffer's cover the basic data types that you can send via IO: byte, short, int, long, float, double and characters.
Java NIO also has a MappedByteBuffer which is used in conjunction with memory mapped files.

Selectors

A Selector allows a single thread to handle multiple Channel's. This is handy if your application has many connections (Channels) open, but only has low traffic on each connection. For instance, in a chat server.
To use a Selector you register the Channel's with it. Then you call it's select() method. This method will block until there is an event ready for one of the registered channels. Once the method returns, the thread can then process these events. Examples of events are incoming connection, data received etc.

Channel

4

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt",
"rw");
FileChannel inChannel = aFile.getChannel();
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {
  System.out.println("Read " + bytesRead);
  buf.flip();
  while(buf.hasRemaining()){
    System.out.print((char) buf.get());
  }
  buf.clear();
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

Industrieel Ingenieur Informatica, UGent

Java NIO Channels are similar to streams with a few differences:
You can both read and write to a Channels. Streams are typically one-way (read or write).
Channels can be read and written asynchronously.
Channels always read to, or write from, a Buffer.
As mentioned above, you read data from a channel into a buffer, and write data from a buffer into a channel.

Channel Implementations
Here are the most important Channel implementations in Java NIO:

FileChannel reads data from and to files.
DatagramChannel can read and write data over the network via UDP.
SocketChannel can read and write data over the network via TCP.
ServerSocketChannel allows you to listen for incoming TCP connections, like a web server does. For each incoming connection a SocketChannel is created.

The **Basic Channel Example**
Notice the buf.flip() call.
First you read into a Buffer. Then you flip it. Then you read out of it.

Instances of **RandomAccessFile** support both reading and writing to a random access file. A random access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the *file pointer*; input

operations read bytes starting at the file pointer and advance the file pointer past the bytes read. If the random access file is created in read/write mode, then output operations are also available; output operations write bytes starting at the file pointer and advance the file pointer past the bytes written. Output operations that write past the current end of the implied array cause the array to be extended. The file pointer can be read by the getFilePointer method and set by the seek method.

Java NIO Buffers are used when interacting with NIO Channels. As you know, data is read from channels into buffers, and written from buffers into channels.
A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.

Basic Buffer Usage
Using a Buffer to read and write data typically follows this little 4-step process:

Write data into the Buffer
Call buffer.flip()
Read data out of the Buffer
Call buffer.clear() or buffer.compact()

When you write data into a buffer, the buffer keeps track of how much data you have written. Once you need to read the data, you need to switch the buffer from writing mode into reading mode using the flip() method call. In reading mode the buffer lets you read all the data written into the buffer.
Once you have read all the data, you need to clear the buffer, to make it ready for writing again. You can do this in two ways: By calling **clear**() or by calling **compact**(). The clear() method clears the whole buffer. The compact() method only clears the data which you have already read. Any unread data is moved to the beginning of the buffer, and data will now be written into the buffer after the unread data.

Allocating a Buffer
To obtain a Buffer object you must first allocate it. Every Buffer class has an allocate() method that does this. Here is an example showing the allocation of a ByteBuffer, with a capacity of 48 bytes:
ByteBuffer buf = ByteBuffer.allocate(48);
Here is an example allocating a CharBuffer with space for 1024 characters:
CharBuffer buf = CharBuffer.allocate(1024);

Writing Data to a Buffer
You can write data into a Buffer in two ways:

Write data from a Channel into a Buffer
Write data into the Buffer yourself, via the buffer's **put**() methods.

Here is an example showing how a Channel can write data into a Buffer:
int bytesRead = inChannel.read(buf); //read into buffer.
Here is an example that writes data into a Buffer via the put() method:
buf.put(127);

There are many other versions of the put() method, allowing you to write data into the Buffer in many different ways. For instance, writing at specific positions, or writing an array of bytes into the buffer. See the JavaDoc for the concrete buffer implementation for more details.

flip()

The flip() method switches a Buffer from writing mode to reading mode. Calling flip() sets the position back to 0, and sets the limit to where position just was.
In other words, position now marks the reading position, and limit marks how many bytes, chars etc. were written into the buffer - the limit of how many bytes, chars etc. that can be read.

Reading Data from a Buffer

There are two ways you can read data from a Buffer.

Read data from the buffer into a channel.

Read data from the buffer yourself, using one of the **get**() methods.

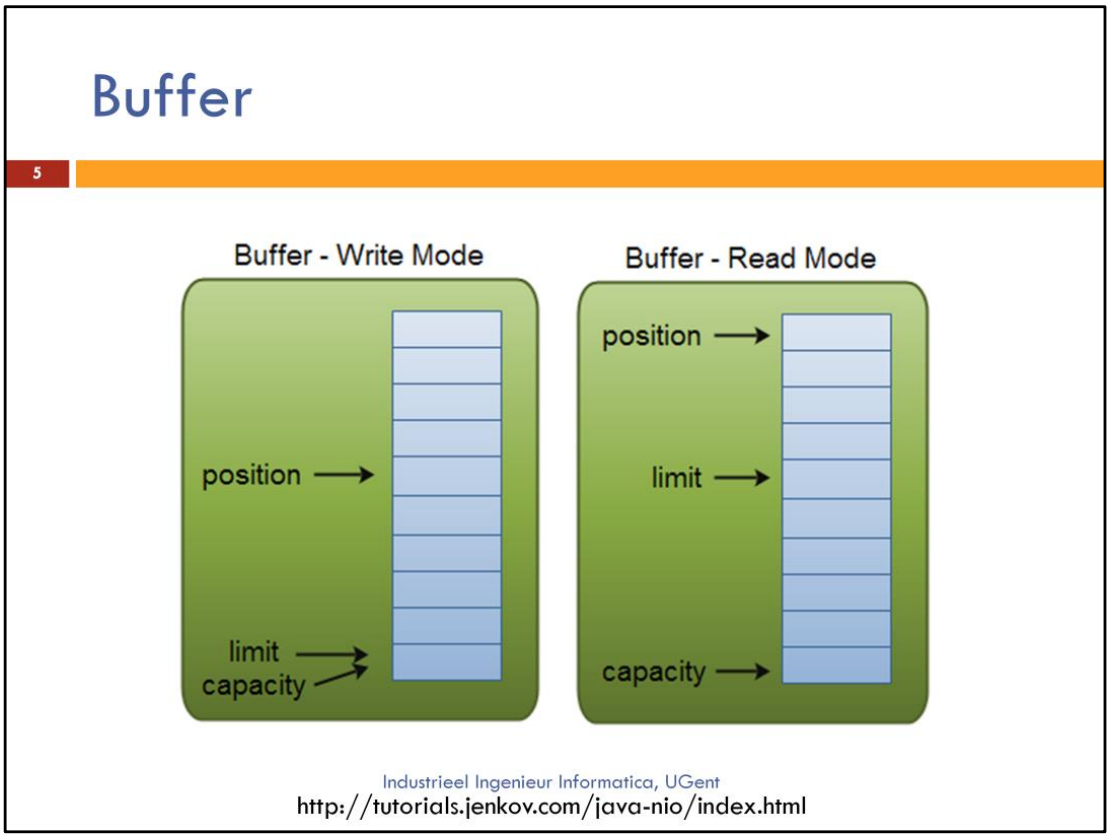
Here is an example of how you can read data from a buffer into a channel:

```
//read from buffer into channel.
int bytesWritten = inChannel.write(buf);
```

Here is an example that reads data from a Buffer using the get() method:

```
byte aByte = buf.get();
```

There are many other versions of the get() method, allowing you to read data from the Buffer in many different ways. For instance, reading at specific positions, or reading an array of bytes from the buffer. See the JavaDoc for the concrete buffer implementation for more details.



Buffer Capacity, Position and Limit

A buffer is essentially a block of memory into which you can write data, which you can then later read again. This memory block is wrapped in a NIO Buffer object, which provides a set of methods that makes it easier to work with the memory block.

A Buffer has three properties you need to be familiar with, in order to understand how a Buffer works. These are:

capacity
position
limit

The meaning of position and limit depends on whether the Buffer is in read or write mode. Capacity always means the same, no matter the buffer mode.

Capacity

Being a memory block, a Buffer has a certain fixed size, also called its "capacity". You can only write capacity bytes, longs, chars etc. into the Buffer. Once the Buffer is full, you need to empty it (read the data, or clear it) before you can write more data into it.

Position

When you write data into the Buffer, you do so at a certain position. Initially the position is 0. When a byte, long etc. has been written into the Buffer the position is advanced to point to the next cell in the buffer to insert data into. Position can maximally become capacity - 1.

When you read data from a Buffer you also do so from a given position. When you flip a Buffer from writing mode to reading mode, the position is reset back to 0. As you read data from the Buffer you do so from position, and position is advanced to next position to read.

Limit

In write mode the limit of a Buffer is the limit of how much data you can write into the buffer. In write mode the limit is equal to the capacity of the Buffer.

When flipping the Buffer into read mode, limit means the limit of how much data you can read from the data. Therefore, when flipping a Buffer into read mode, limit is set to write position of the write mode. In other words, you can read as many bytes as were written (limit is set to the number of bytes written, which is marked by position).

Java NIO comes with the following **Buffer** types:

- ByteBuffer
- MappedByteBuffer
- CharBuffer
- DoubleBuffer
- FloatBuffer
- IntBuffer
- LongBuffer
- ShortBuffer

As you can see, these Buffer types represent different data types. In other words, they let you work with the bytes in the buffer as char, short, int, long, float or double instead.

MappedByteBuffer: a direct byte buffer whose content is a memory-mapped region of a file.

rewind()

The Buffer.rewind() sets the position back to 0, so you can reread all the data in the buffer. The limit remains untouched, thus still marking how many elements (bytes, chars etc.) that can be read from the Buffer.

clear() and compact()

Once you are done reading data out of the Buffer you have to make the Buffer ready for writing again. You can do so either by calling clear() or by calling compact().

If you call clear() the position is set back to 0 and the limit to capacity. In other words, the Buffer is cleared. The data in the Buffer is not cleared. Only the markers telling where you can write data into the Buffer are.

If there is any unread data in the Buffer when you call clear() that data will be "forgotten", meaning you no longer have any markers telling what data has been read, and what has not been read.

If there is still unread data in the Buffer, and you want to read it later, but you need to do some writing first, call compact() instead of clear().

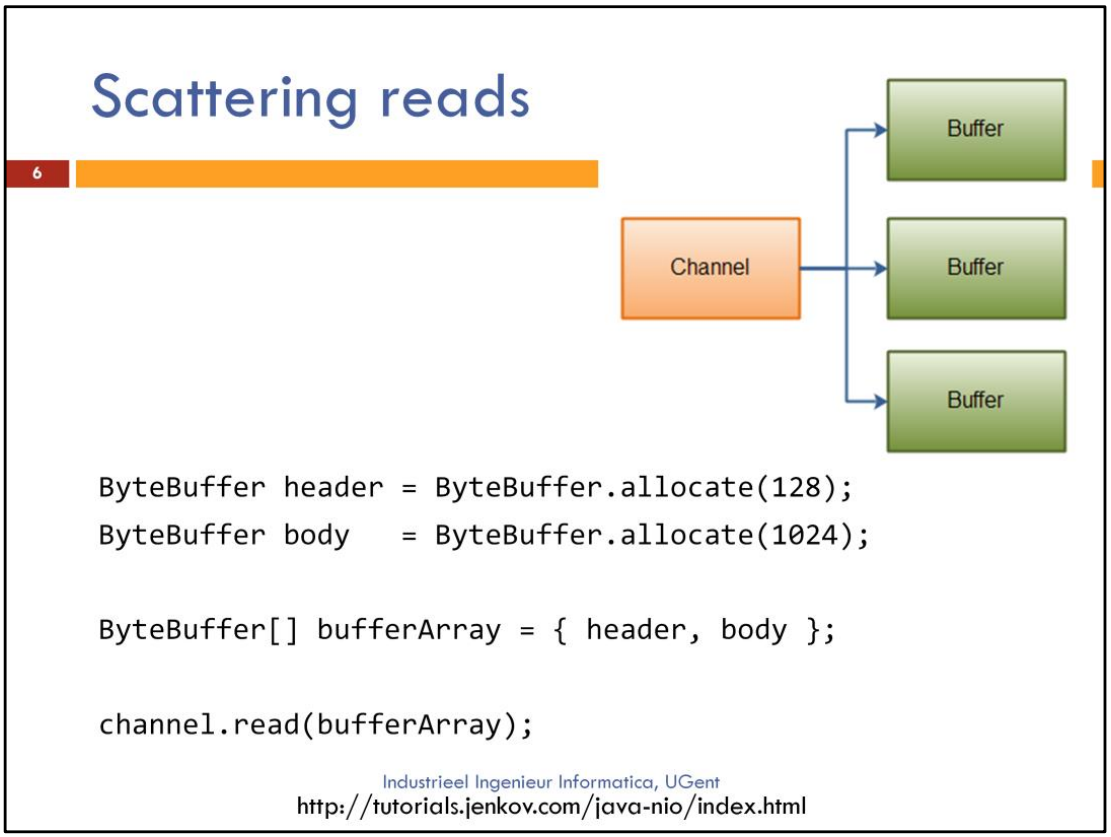
compact() copies all unread data to the beginning of the Buffer. Then it sets position

to right after the last unread element. The limit property is still set to capacity, just like clear() does. Now the Buffer is ready for writing, but you will not overwrite the unread data.

mark() and reset()
You can mark a given position in a Buffer by calling the Buffer.mark() method. You can then later reset the position back to the marked position by calling the Buffer.reset() method. Here is an example:

```
buffer.mark();
//call buffer.get() a couple of times, e.g. during parsing.
buffer.reset(); //set position back to mark.
```

equals() and compareTo()
It is possible to compare two buffers using equals() and compareTo().
equals()
Two buffers are equal if:
They are of the same type (byte, char, int etc.)
They have the same amount of remaining bytes, chars etc. in the buffer.
All remaining bytes, chars etc. are equal.
As you can see, equals only compares part of the Buffer, not every single element inside it. In fact, it just compares the remaining elements in the Buffer.
compareTo()
The compareTo() method compares the remaining elements (bytes, chars etc.) of the two buffers, for use in e.g. sorting routines. A buffer is considered "smaller" than another buffer if:
The first element which is equal to the corresponding element in the other buffer, is smaller than that in the other buffer.
All elements are equal, but the first buffer runs out of elements before the second buffer does (it has fewer elements).

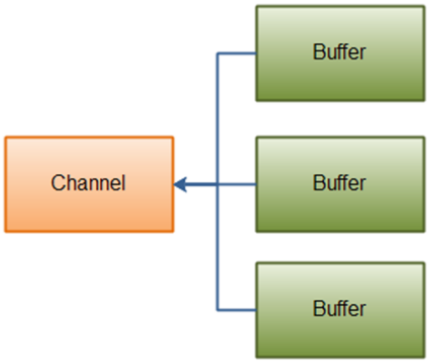


Java NIO comes with built-in scatter / gather support. Scatter / gather are concepts used in reading from, and writing to channels.
A scattering read from a channel is a read operation that reads data into more than one buffer. Thus, the channel "scatters" the data from the channel into multiple buffers.
A gathering write to a channel is a write operation that writes data from more than one buffer into a single channel. Thus, the channel "gathers" the data from multiple buffers into one channel.
Scatter / gather can be really useful in situations where you need to work with various parts of the transmitted data separately. For instance, if a message consists of a header and a body, you might keep the header and body in separate buffers. Doing so may make it easier for you to work with header and body separately.
Scattering Reads
A "scattering read" reads data from a single channel into multiple buffers.

Notice how the buffers are first inserted into an array, then the array passed as parameter to the channel.read() method. The read() method then writes data from the channel in the sequence the buffers occur in the array. Once a buffer is full, the channel moves on to fill the next buffer.
The fact that scattering reads fill up one buffer before moving on to the next, means that it is not suited for dynamically sized message parts. In other words, if you have a header and a body, and the header is fixed size (e.g. 128 bytes), then a scattering read works fine.

Gathering writes

7




```
ByteBuffer header = ByteBuffer.allocate(128);
ByteBuffer body   = ByteBuffer.allocate(1024);

//write data into buffers

ByteBuffer[] bufferArray = { header, body };

channel.write(bufferArray);
```

Industrieel Ingenieur Informatica, UGent
http://tutorials.jenkov.com/java-nio/index.html

A "gathering write" writes data from multiple buffers into a single channel.

The array of buffers are passed into the write() method, which writes the content of the buffers in the sequence they are encountered in the array. Only the data between position and limit of the buffers is written. Thus, if a buffer has a capacity of 128 bytes, but only contains 58 bytes, only 58 bytes are written from that buffer to the channel. Thus, a gathering write works fine with dynamically sized message parts, in contrast to scattering reads.

Channel to channel transfer

8

```
RandomAccessFile fromFile
  = new RandomAccessFile("fromFile.txt", "rw");
FileChannel fromChannel = fromFile.getChannel();

RandomAccessFile toFile
  = new RandomAccessFile("toFile.txt", "rw");
FileChannel toChannel = toFile.getChannel();

long position = 0;
long count = fromChannel.size();

toChannel.transferFrom(fromChannel, position, count);
```

Industrieel Ingenieur Informatica, UGent

In Java NIO you can transfer data directly from one channel to another, if one of the channels is a FileChannel. The **FileChannel** class has a **transferTo**() and a **transferFrom**() method which does this for you.

transferFrom()
The FileChannel.transferFrom() method transfers data from a source channel into the FileChannel.

The parameters position and count, tell where in the destination file to start writing (position), and how many bytes to transfer maximally (count). If the source channel has fewer than count bytes, less is transfered.
Additionally, some SocketChannel implementations may transfer only the data the SocketChannel has ready in its internal buffer here and now - even if the SocketChannel may later have more data available. Thus, it may not transfer the entire data requested (count) from the SocketChannel into FileChannel.

Channel to channel transfer

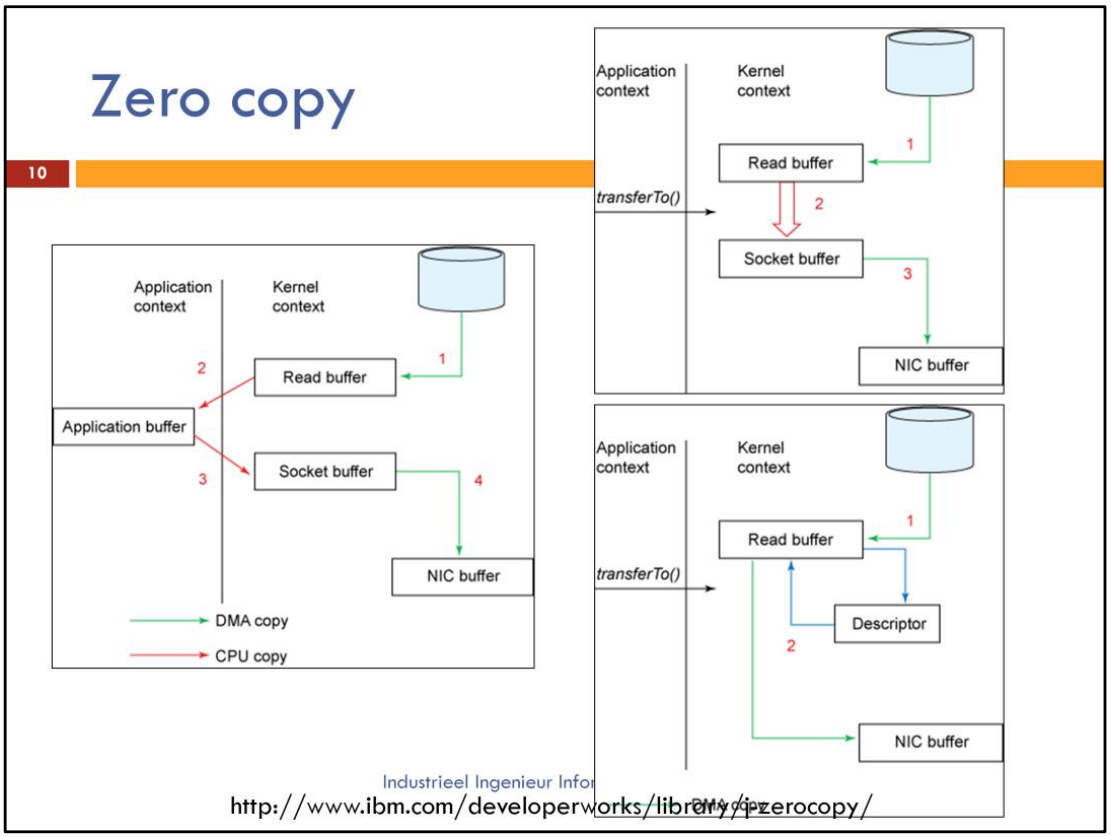
9

```
RandomAccessFile fromFile
  = new RandomAccessFile("fromFile.txt", "rw");
FileChannel fromChannel = fromFile.getChannel();

RandomAccessFile toFile
  = new RandomAccessFile("toFile.txt", "rw");
FileChannel toChannel = toFile.getChannel();

long position = 0;
long count    = fromChannel.size();

fromChannel.transferTo(position, count, toChannel);
```

Industrieel Ingenieur Informatica, UGent

transferTo()
The transferTo() method transfer from a FileChannel into some other channel.

Notice how similar the example is to the previous. The only real difference is the which FileChannel object the method is called on. The rest is the same.
The issue with SocketChannel is also present with the transferTo() method. The SocketChannel implementation may only transfer bytes from the FileChannel until the send buffer is full, and then stop.



Many Web applications serve a significant amount of static content, which amounts to reading data off of a disk and writing the exact same data back to the response socket. This activity might appear to require relatively little CPU activity, but it's somewhat inefficient: the kernel reads the data off of disk and pushes it across the kernel-user boundary to the application, and then the application pushes it back across the kernel-user boundary to be written out to the socket. In effect, the application serves as an inefficient intermediary that gets the data from the disk file to the socket.

Each time data traverses the user-kernel boundary, it must be copied, which consumes CPU cycles and memory bandwidth. Fortunately, you can eliminate these copies through a technique called — appropriately enough — *zero copy*. Applications that use zero copy request that the kernel copy the data directly from the disk file to the socket, without going through the application. Zero copy greatly improves application performance and reduces the number of context switches between kernel and user mode.

The Java class libraries support zero copy on Linux and UNIX systems through the transferTo() method in java.nio.channels.FileChannel. You can use the transferTo() method to transfer bytes directly from the channel on which it is invoked to another writable byte channel, without requiring data to flow through the application. This article first demonstrates the overhead incurred by simple file transfer done through traditional copy semantics, then shows how the zero-copy technique using transferTo() achieves better performance.

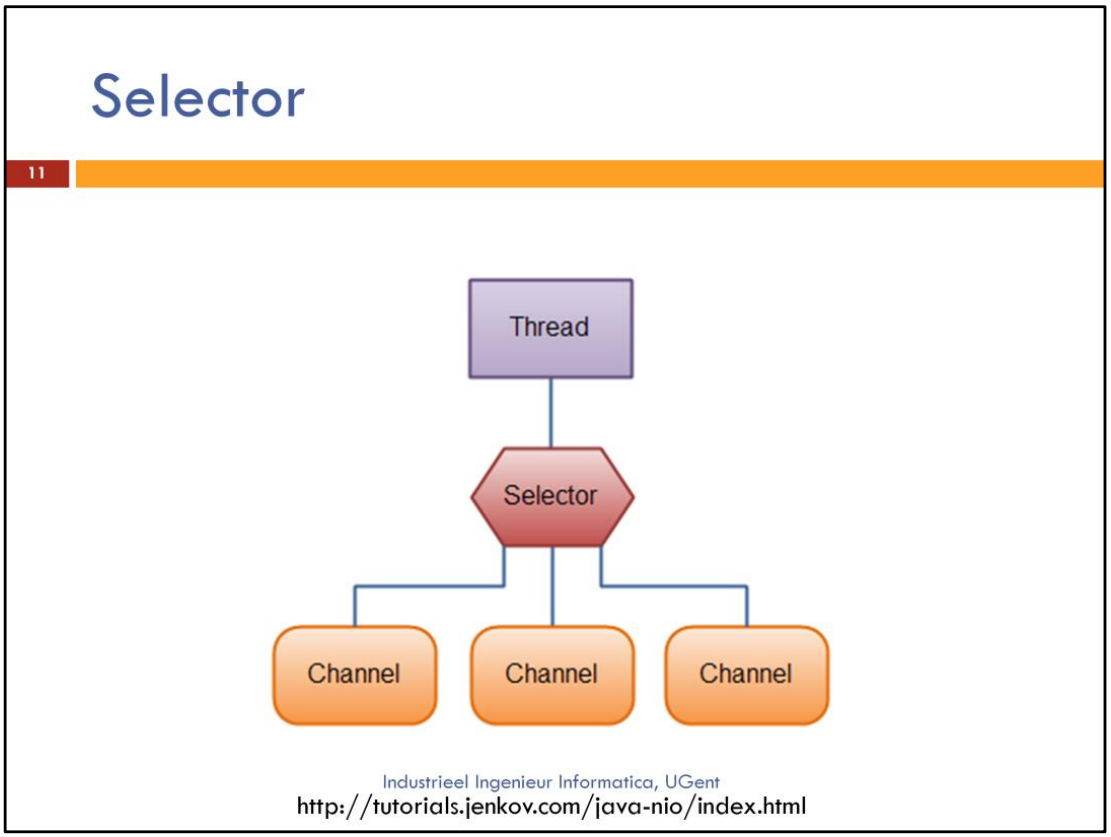
Date transfer: The traditional approach
Consider the scenario of reading from a file and transferring the data to another program over the network. (This scenario describes the behavior of many server applications, including Web applications serving static content, FTP servers, mail servers, and so on.) (figuur 1)

Use of the intermediate kernel buffer (rather than a direct transfer of the data into the user buffer) might seem inefficient. But intermediate kernel buffers were introduced into the process to improve performance. Using the intermediate buffer on the read side allows the kernel buffer to act as a "readahead cache" when the application hasn't asked for as much data as the kernel buffer holds. This significantly improves performance when the requested data amount is less than the kernel buffer size. The intermediate buffer on the write side allows the write to complete asynchronously.
Unfortunately, this approach itself can become a performance bottleneck if the size of the data requested is considerably larger than the kernel buffer size. The data gets copied multiple times among the disk, kernel buffer, and user buffer before it is finally delivered to the application.
Zero copy improves performance by eliminating these redundant data copies.

Data transfer: The zero-copy approach
If you re-examine the traditional scenario, you'll notice that the second and third data copies are not actually required. The application does nothing other than cache the data and transfer it back to the socket buffer. Instead, the data could be transferred directly from the read buffer to the socket buffer. The transferTo() method lets you do exactly this. (figuur 2)

This is an improvement: we've reduced the number of context switches from four to two and reduced the number of data copies from four to three (only one of which involves the CPU). But this does not yet get us to our goal of zero copy. We can further reduce the data duplication done by the kernel if the underlying network interface card supports *gather operations*. In Linux kernels 2.4 and later, the socket buffer descriptor was modified to accommodate this requirement. This approach not only reduces multiple context switches but also eliminates the duplicated data copies that require CPU involvement. The user-side usage still remains the same, but the intrinsics have changed:

> The transferTo() method causes the file contents to be copied into a kernel buffer by the DMA (direct memory access) engine.
> No data is copied into the socket buffer. Instead, only descriptors with information about the location and length of the data are appended to the socket buffer. The DMA engine passes data directly from the kernel buffer to the protocol engine, thus eliminating the remaining final CPU copy.



A Selector is a Java NIO component which can examine one or more NIO Channel's, and determine which channels are ready for e.g. reading or writing. This way a single thread can manage multiple channels, and thus multiple network connections.

Why Use a Selector?

The advantage of using just a single thread to handle multiple channels is that you need less threads to handle the channels. Actually, you can use just one thread to handle all of your channels. Switching between threads is expensive for an operating system, and each thread takes up some resources (memory) in the operating system too. Therefore, the less threads you use, the better.

Keep in mind though, that modern operating systems and CPU's become better and better at multitasking, so the overheads of multithreading becomes smaller over time. In fact, if a CPU has multiple cores, you might be wasting CPU power by **not** multitasking. Anyways, that design discussion belongs in a different text. It suffices to say here, that you can handle multiple channels with a single thread, using a Selector.

Selector – aanmaak en registratie

12

```
// aanmaak selector
Selector selector = Selector.open();

// registratie één of meerdere kanalen
channel.configureBlocking(false);
SelectionKey key
  = channel.register(selector, SelectionKey.OP_READ);
```

Industrieel Ingenieur Informatica, UGent

Creating a Selector
You create a Selector by calling the Selector.open() method, like this:
Selector selector = Selector.open();

Registering Channels with the Selector
In order to use a Channel with a Selector you must register the Channel with the Selector. This is done using the SelectableChannel.register() method, like this:
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
The Channel must be in non-blocking mode to be used with a Selector. This means that you cannot use FileChannel's with a Selector since FileChannel's cannot be switched into non-blocking mode. Socket channels will work fine though.
Notice the second parameter of the register() method. This is an "interest set", meaning what events you are interested in listening for in the Channel, via the Selector. There are four different events you can listen for:

Connect
Accept
Read
Write

A channel that "fires an event" is also said to be "ready" for that event. So, a channel that has connected successfully to another server is "connect ready". A server socket channel which accepts an incoming connection is "accept" ready. A channel that has data ready to be read is "read" ready. A channel that is ready for you to write data to

it, is "write" ready.
These four events are represented by the four SelectionKey constants:

```
SelectionKey.OP_CONNECT
SelectionKey.OP_ACCEPT
SelectionKey.OP_READ
SelectionKey.OP_WRITE
```

If you are interested in more than one event, OR the constants together, like this:

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

Selector - gebruik

13

```
while(true) {
  // wachten
  int readyChannels = selector.select();
  if(readyChannels == 0) continue;

  for (SelectionKey key : selector.selectedKeys()) {
    .. // kanaal klaar om te …
  }
  selector.selectedKeys().clear();
}
```

Industrieel Ingenieur Informatica, UGent

Selecting Channels via a Selector
Once you have register one or more channels with a Selector you can call one of the **select**() methods. These methods return the channels that are "ready" for the events you are interested in (connect, accept, read or write). In other words, if you are interested in channels that are ready for reading, you will receive the channels that are ready for reading from the select() methods.

Here are the select() methods:
int select() blocks until at least one channel is ready for the events you registered for.
int select(long timeout) does the same as select() except it blocks for a maximum of timeout milliseconds (the parameter).
int selectNow() doesn't block at all. It returns immediately with whatever channels are ready.

The int returned by the select() methods tells how many channels are ready. That is, how many channels that became ready since last time you called select(). If you call select() and it returns 1 because one channel has become ready, and you call select() one more time, and one more channel has become ready, it will return 1 again. If you have done nothing with the first channel that was ready, you now have 2 ready channels, but only one channel had become ready between each select() call.

selectedKeys()
Once you have called one of the select() methods and its return value has indicated

that one or more channels are ready, you can access the ready channels via the "selected key set", by calling the selectors selectedKeys() method.

When you register a channel with a Selector the Channel.register() method returns a SelectionKey object. This key represents that channels registration with that selector. It is these keys you can access via the selectedKeySet() method.

You can iterate this selected key set to access the ready channels.
This loop iterates the keys in the selected key set. For each key it tests the key to determine what the channel referenced by the key is ready for.

Notice the `selector.selectedKeys().clear();` call after the iteration. The Selector does not remove the SelectionKey instances from the selected key set itself. You have to do this, when you are done processing the channel. The next time the channel becomes "ready" the Selector will add it to the selected key set again.

The channel returned by the SelectionKey.channel() method should be cast to the channel you need to work with, e.g a ServerSocketChannel or SocketChannel etc.

wakeUp()
A thread that has called the select() method which is blocked, can be made to leave the select() method, even if no channels are yet ready. This is done by having a different thread call the Selector.wakeup() method on the Selector which the first thread has called select() on. The thread waiting inside select() will then return immediately.
If a different thread calls wakeup() and no thread is currently blocked inside select(), the next thread that calls select() will "wake up" immediately.
close()
When you are finished with the Selector you call its close() method. This closes the Selector and invalidates all SelectionKey instances registered with this Selector. The channels themselves are not closed.

Selector - communicatie

14

```
if(key.isAcceptable()) {
    // a connection was accepted by a ServerSocketChannel.
    Channel channel = key.channel(); …
} else if (key.isConnectable()) {
    // a connection was established with a remote server.
} else if (key.isReadable()) {
    // a channel is ready for reading
} else if (key.isWritable()) {
    // a channel is ready for writing
}
…
```

Industrieel Ingenieur Informatica, UGent

SelectionKey's
As you saw in the previous section, when you register a Channel with a Selector the register() method returns a SelectionKey objects. This SelectionKey object contains a few interesting properties:

The interest set
The ready set
The Channel
The Selector
An attached object (optional)

Interest Set
The interest set is the set of events you are interested in "selecting", as described in the section "Registering Channels with the Selector". You can read and write that interest set via the SelectionKey like this:

```
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept = interestSet & SelectionKey.OP_ACCEPT;
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInRead = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite = interestSet & SelectionKey.OP_WRITE;
```

As you can see, you can AND the interest set with the given SelectionKey constant to find out if a certain event is in the interest set.

Ready Set
The ready set is the set of operations the channel is ready for. You will primarily be accessing the ready set after a selection. You access the ready set like this:
int readySet = selectionKey.readyOps();
You can test in the same way as with the interest set, what events / operations the channel is ready for. But, you can also use these four methods instead, which all return a boolean:

```
selectionKey.isAcceptable();
selectionKey.isConnectable();
selectionKey.isReadable();
selectionKey.isWritable();
```

Channel + Selector
Accessing the channel + selector from the SelectionKey is trivial. Here is how it's done:

```
Channel channel = selectionKey.channel();
Selector selector = selectionKey.selector();
```

Attaching Objects
You can attach an object to a SelectionKey this is a handy way of recognizing a given channel, or attaching further information to the channel. For instance, you may attach the Buffer you are using with the channel, or an object containing more aggregate data. Here is how you attach objects:

```
selectionKey.attach(theObject);
Object attachedObj = selectionKey.attachment();
```

You can also attach an object already while registering the Channel with the Selector, in the register() method. Here is how that looks:
SelectionKey key = channel.register(selector, SelectionKey.OP_READ, theObject);

The channel returned by the SelectionKey.channel() method should be cast to the channel you need to work with, e.g a ServerSocketChannel or SocketChannel etc.

wakeUp()
A thread that has called the select() method which is blocked, can be made to leave the select() method, even if no channels are yet ready. This is done by having a different thread call the Selector.wakeup() method on the Selector which the first thread has called select() on. The thread waiting inside select() will then return immediately.
If a different thread calls wakeup() and no thread is currently blocked inside select(), the next thread that calls select() will "wake up" immediately.

close()
When you are finished with the Selector you call its close() method. This closes the Selector and invalidates all SelectionKey instances registered with this Selector. The channels themselves are not closed.

SocketChannel: aanmaak - lezen

15

```
// aanmaken
SocketChannel socketChannel = SocketChannel.open();
socketChannel.connect(
  new InetSocketAddress("http://jenkov.com", 80));

// lezen
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = socketChannel.read(buf);
```

Industrieel Ingenieur Informatica, UGent

A Java NIO SocketChannel is a channel that is connected to a TCP network socket. It is Java NIO's equivalent of Java Networking's Sockets. There are two ways a SocketChannel can be created:

- You open a SocketChannel and connect to a server somewhere on the internet.
- A SocketChannel can be created when an incoming connection arrives at a ServerSocketChannel.

Reading from a SocketChannel

First a Buffer is allocated. The data read from the SocketChannel is read into the Buffer.

Second the SocketChannel.read() method is called. This method reads data from the SocketChannel into the Buffer. The int returned by the read() method tells how many bytes were witten into the Buffer. If -1 is returned, the end-of-stream is reached (the connection is closed).

SocketChannel: schrijven - sluiten

16

```
// buffer met info
String newData = "New String to write to file...";
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
// data schrijven
buf.flip();
while(buf.hasRemaining()) {
    socketChannel.write(buf);
}
// afsluiten
socketChannel.close();
```

Industrieel Ingenieur Informatica, UGent

Writing to a SocketChannel
Notice how the SocketChannel.write() method is called inside a while-loop. There is no guarantee of how many bytes the write() method writes to the SocketChannel. Therefore we repeat the write() call until the Buffer has no further bytes to write.

Non-blocking Mode
You can set a SocketChannel into non-blocking mode. When you do so, you can call connect(), read() and write() in asynchronous mode.

If the SocketChannel is in non-blocking mode, and you call connect(), the method may return before a connection is established. To determine whether the connection is established, you can call the finishConnect() method.

In non-blocking mode the write() method may return without having written anything.

In non-blocking mode the read() method may return without having read any data at all. Therefore you need to pay attention to the returned int, which tells how many bytes were read.

The non-blocking mode of SocketChannel's works best with Selector's. By registering one or more SocketChannel's with a Selector, you can ask the Selector for channels that are ready for reading, writing etc.

ServerSocketChannel: aanmaak

17

```
ServerSocketChannel serverSocketChannel =
  ServerSocketChannel.open();

serverSocketChannel.socket().bind(new InetSocketAddress(9999));

while(true){
  SocketChannel socketChannel = serverSocketChannel.accept();

  //do something with socketChannel...
}

serverSocketChannel.close();
```

Industrieel Ingenieur Informatica, UGent

Non-blocking Mode

A ServerSocketChannel can be set into non-blocking mode. In non-blocking mode the accept() method returns immediately, and may thus return null, if no incoming connection had arrived. Therefore you will have to check if the returned SocketChannel is null.

Voorbeeld

- ChatServerProgram.java
- ChatServer.java

Java IO versus java NIO

19

IO	NIO
Stream oriented	Buffer oriented
Blocking IO	Non blocking IO
	Selectors

Industrieel Ingenieur Informatica, UGent

When studying both the Java NIO and IO API's, a question quickly pops into mind: When should I use IO and when should I use NIO?

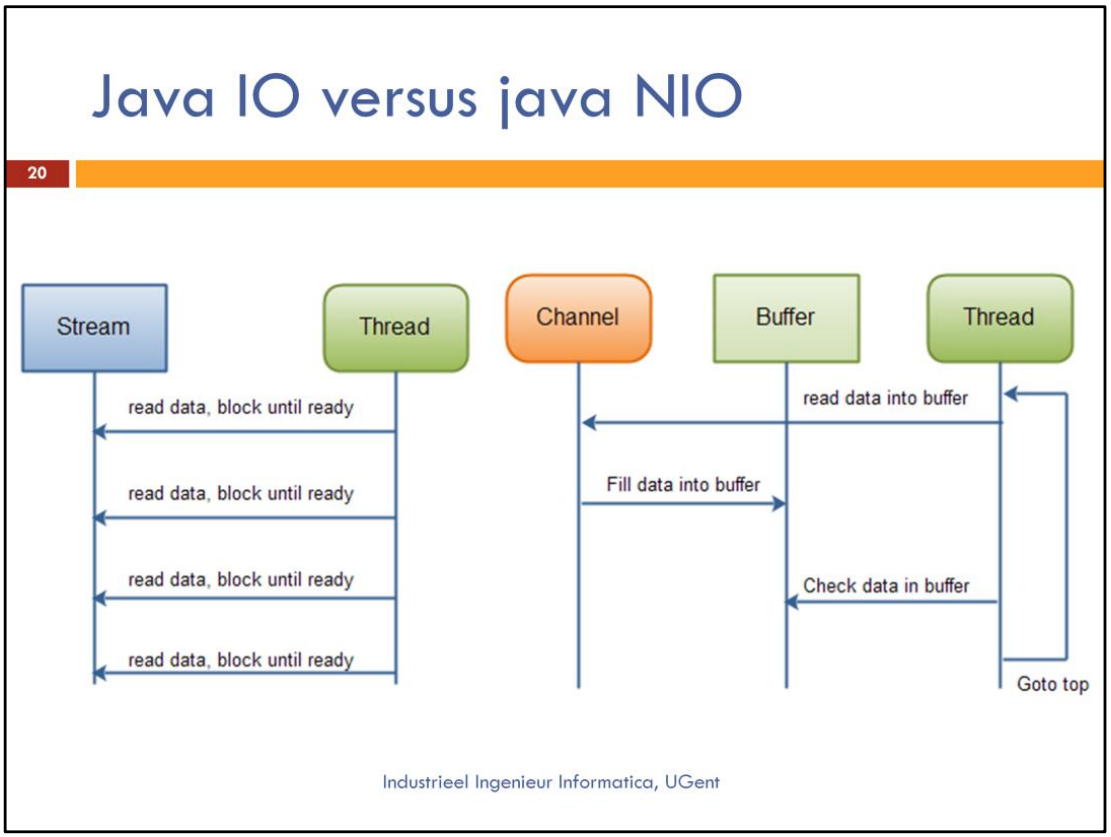
Main Differences Between Java NIO and IO
The table above summarizes the main differences between Java NIO and IO.

Stream Oriented vs. Buffer Oriented
The first big difference between Java NIO and IO is that IO is stream oriented, where NIO is buffer oriented. So, what does that mean?
Java IO being stream oriented means that you read one or more bytes at a time, from a stream. What you do with the read bytes is up to you. They are not cached anywhere. Furthermore, you cannot move forth and back in the data in a stream. If you need to move forth and back in the data read from a stream, you will need to cache it in a buffer first.
Java NIO's buffer oriented approach is slightly different. Data is read into a buffer from which it is later processed. You can move forth and back in the buffer as you need to. This gives you a bit more flexibility during processing. However, you also need to check if the buffer contains all the data you need in order to fully process it. And, you need to make sure that when reading more data into the buffer, you do not overwrite data in the buffer you have not yet processed.

Blocking vs. Non-blocking IO
Java IO's various streams are blocking. That means, that when a thread invokes a

read() or write(), that thread is blocked until there is some data to read, or the data is fully written. The thread can do nothing else in the meantime.
Java NIO's non-blocking mode enables a thread to request reading data from a channel, and only get what is currently available, or nothing at all, if no data is currently available. Rather than remain blocked until data becomes available for reading, the thread can go on with something else.
The same is true for non-blocking writing. A thread can request that some data be written to a channel, but not wait for it to be fully written. The thread can then go on and do something else in the mean time.
What threads spend their idle time on when not blocked in IO calls, is usually performing IO on other channels in the meantime. That is, a single thread can now manage multiple channels of input and output.

Selectors
Java NIO's selectors allow a single thread to monitor multiple channels of input. You can register multiple channels with a selector, then use a single thread to "select" the channels that have input available for processing, or select the channels that are ready for writing. This selector mechanism makes it easy for a single thread to manage multiple channels.



How NIO and IO Influences Application Design
Whether you choose NIO or IO as your IO toolkit may impact the following aspects of your application design:

The API calls to the NIO or IO classes.
The processing of data.
The number of thread used to process the data.

The API Calls
Of course the API calls when using NIO look different than when using IO. This is no surprise. Rather than just read the data byte for byte from e.g. an InputStream, the data must first be read into a buffer, and then be processed from there.

The Processing of Data
The processing of the data is also affected when using a pure NIO design, vs. an IO design.

In an **IO** design you read the data byte for byte from an InputStream or a Reader. Imagine you were processing a stream of line based textual data. For instance:
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890

This stream of text lines could be processed like this:

```
InputStream input = ... ; // get the InputStream from the client socket
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String nameLine = reader.readLine();
String ageLine = reader.readLine();
String emailLine = reader.readLine();
String phoneLine = reader.readLine();
```

Notice how the processing state is determined by how far the program has executed. In other words, once the first reader.readLine() method returns, you know for sure that a full line of text has been read. The readLine() blocks until a full line is read, that's why. You also know that this line contains the name. Similarly, when the second readLine() call returns, you know that this line contains the age etc.
As you can see, the program progresses only when there is new data to read, and for each step you know what that data is. Once the executing thread have progressed past reading a certain piece of data in the code, the thread is not going backwards in the data (mostly not).

A **NIO implementation** would look different. Here is a simplified example:

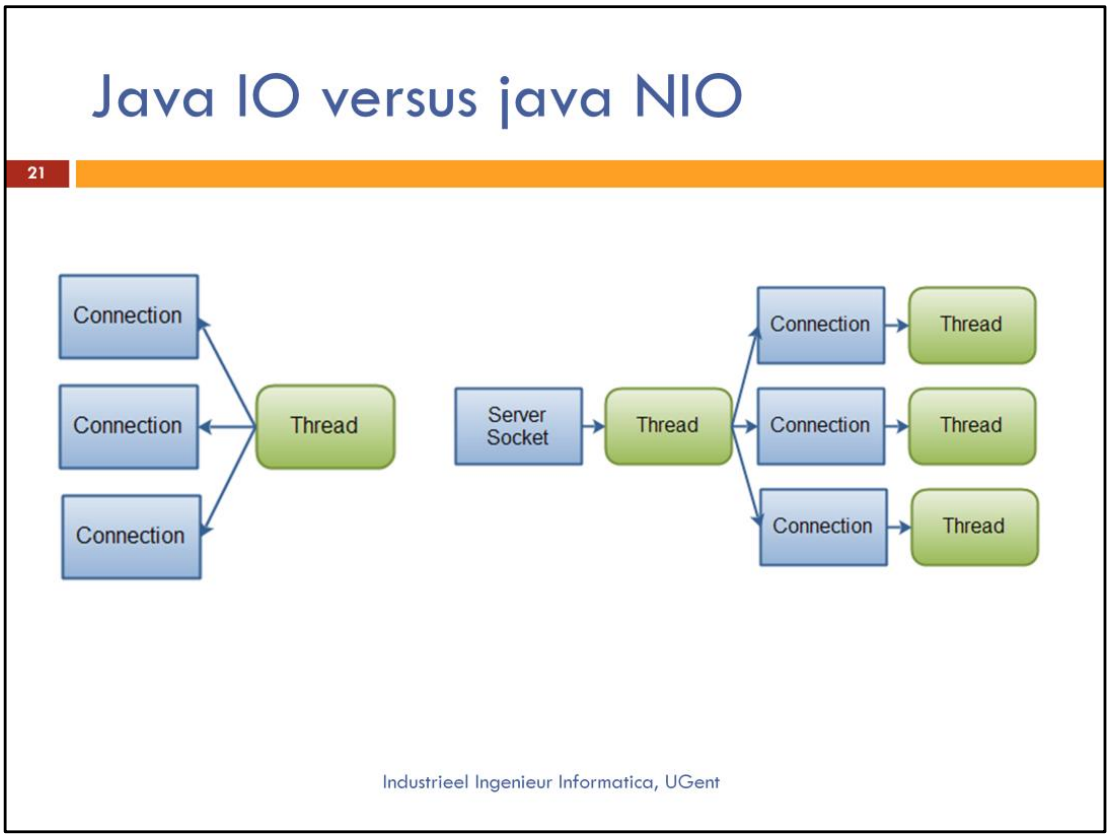
```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
```

Notice the second line which reads bytes from the channel into the ByteBuffer. When that method call returns you don't know if all the data you need is inside the buffer. All you know is that the buffer contains some bytes. This makes processing somewhat harder.
Imagine if, after the first read(buffer) call, that all what was read into the buffer was half a line. For instance, "Name: An". Can you process that data? Not really. You need to wait until at leas a full line of data has been into the buffer, before it makes sense to process any of the data at all.
So how do you know if the buffer contains enough data for it to make sense to be processed? Well, you don't. The only way to find out, is to look at the data in the buffer. The result is, that you may have to inspect the data in the buffer several times before you know if all the data is inthere. This is both inefficient, and can become messy in terms of program design.
For instance:

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while(! bufferFull(bytesRead) ) {
  bytesRead = inChannel.read(buffer);
}
```

The bufferFull() method has to keep track of how much data is read into the buffer, and return either true or false, depending on whether the buffer is full. In other words, if the buffer is ready for processing, it is considered full.
The bufferFull() method scans through the buffer, but must leave the buffer in the

same state as before the bufferFull() method was called. If not, the next data read into the buffer might not be read in at the correct location. This is not impossible, but it is yet another issue to watch out for.
If the buffer is full, it can be processed. If it is not full, you might be able to partially process whatever data is there, if that makes sense in your particular case. In many cases it doesn't.



Summary
NIO allows you to manage multiple channels (network connections or files) using only a single (or few) threads, but the cost is that parsing the data might be somewhat more complicated than when reading data from a blocking stream.
If you need to manage thousands of open connections simultanously, which each only send a little data, for instance a chat server, implementing the server in NIO is probably an advantage. Similarly, if you need to keep a lot of open connections to other computers, e.g. in a P2P network, using a single thread to manage all of your outbound connections might be an advantage.

If you have fewer connections with very high bandwidth, sending a lot of data at a time, perhaps a classic IO server implementation might be the best fit.

The primary contribution of NIO and NIO.2 to the Java platform is to improve performance in one of the core areas of Java application development: input/output processing. Neither package is particularly easy to work with, nor are the New Input/Output APIs required for every Java I/O scenario. Used correctly, though, Java NIO and NIO.2 can slash the time required for some common I/O operations. That's the superpower of NIO and NIO.2, and this article presents five relatively simple ways to leverage it:

Java IO versus java NIO

22

- Change notifiers
- Selectors help multiplex
- Channels
- Memory mapping
- Character encoding and searching

Industrieel Ingenieur Informatica, UGent

http://www.javaworld.com/article/2078654/java-se/java-se-five-ways-to-maximize-java-nio-and-nio-2.html

Change notifiers (because everybody needs a listener)

Many enterprise-class applications need to take a specific action when:

A file is uploaded into an FTP folder
A configuration definition is changed
A draft document is updated
Another file-system event occurs

These are all examples of change notification or change response. In early versions of Java (and other languages), *polling* was typically the best way to detect change events. Polling is a particular kind of endless loop: check the file-system or other object, compare it to its last-known state, and, if there's no change, check back again after a brief interval, such as a hundred milliseconds or ten seconds. Continue the loop indefinitely.

NIO.2 gives us a better way to express change detection. NIO.2's Watcher class is considerably more straightforward and easy-to-use for change notification than the traditional I/O solution based on polling.

Selectors and asynchronous I/O: Selectors help multiplex

Newcomers to NIO sometimes associate it with "non-blocking input/output." NIO is more than non-blocking I/O but the error makes sense: basic I/O in Java is *blocking* -- meaning that it waits until it can complete an operation -- whereas non-blocking, or asynchronous, I/O is one of the most-used NIO facilities.

NIO's non-blocking I/O is *event-based*. This means that a *selector* (or callback or listener) is defined for an I/O channel, then processing continues. When an event

occurs on the selector -- when a line of input arrives, for instance -- the selector "wakes up" and executes. All of this is achieved *within a single thread*, which is a significant contrast to typical Java I/O. Unix and Unix-like operating systems have long had efficient implementations of selectors, so this sort of networking program is a model of good performance for a Java-coded networking program.

Channels: Promise and reality

In NIO, a *channel* can be any object that reads or writes. Its job is to abstract files and sockets. NIO channels support a consistent collection of methods, so it's possible to program without having special cases depending on whether stdout, a network connection, or some other channel is actually in use. Channels share this characteristic of the *streams* of Java's basic I/O. Streams provide blocking I/O; channels support asynchronous I/O.

While NIO is often promoted for its performance advantages, it's more precise to say it is highly *responsive*. In some cases NIO actually performs *worse* than basic Java I/O. For simple sequential reads and writes of small files, for instance, a straightforward streams implementation might be two or three times faster than the corresponding event-oriented channel-based coding. Also, *non*-multiplexed channels -- that is, channels in separate threads -- can be much slower than channels that register their selectors in a single thread.

The next time you need to define a programming problem in terms of dimensions pertinent to streams or channels, try asking the following questions:

> How many I/O objects must you read and write?
> Is there a natural sequence between different I/O objects or might they all need to happen simultaneously?
> Do your I/O objects only last for a short interval or are they likely to persist during the lifetime of your process?
> Is it more natural to do your I/O in a single thread or several distinct ones?
> Is network traffic likely to look the same as local I/O or do the two have different patterns?

This sort of analysis is good practice for deciding when to use streams or channels. Remember: NIO and NIO.2 don't replace basic I/O; they just supplement it.

Memory mapping -- where it counts

The most consistently dramatic performance improvement in the use of NIO involves memory mapping. *Memory mapping* is an OS-level service that makes segments of a file appear for programming purposes like areas of memory. Memory mapping has a number of consequences and implications, more than I'll get into here. At a high level, it helps make I/O happen at the speed of memory access, rather than file access. The former is often two orders of magnitude faster than the latter.

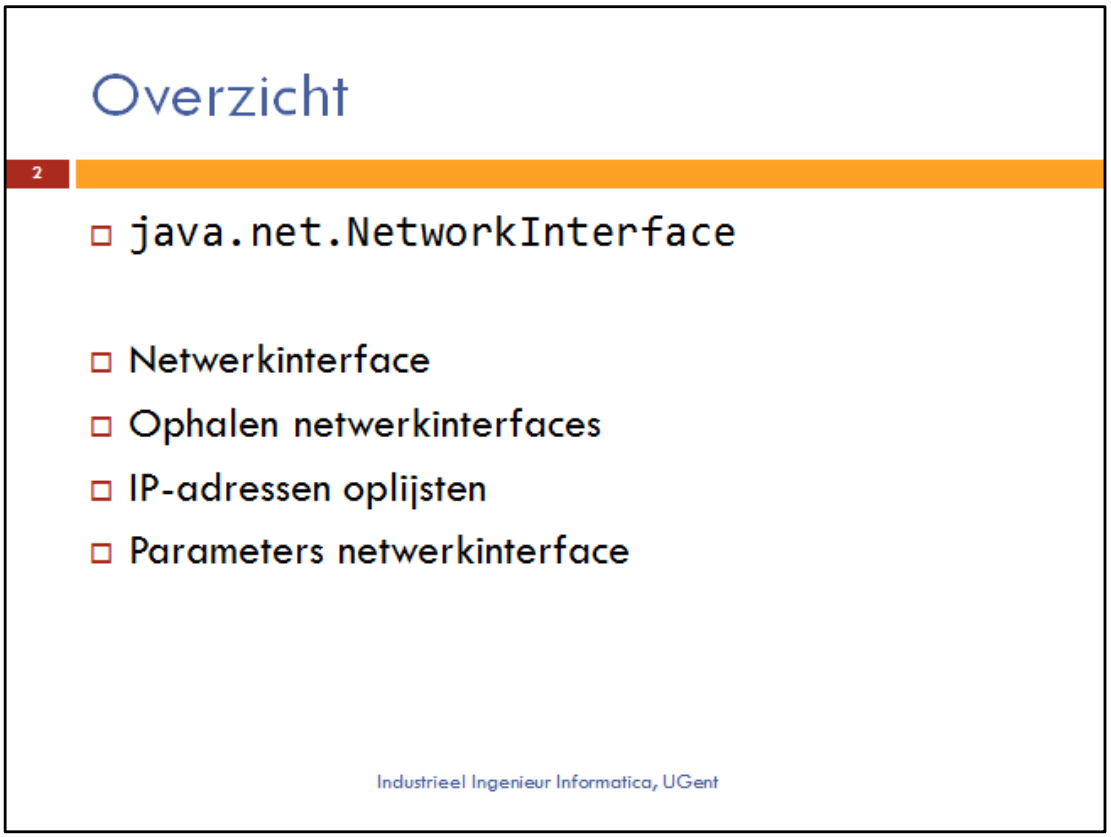
In practical cases, memory mapping is interesting not only for the raw speed of I/O, but also because several different readers and writers can attach simultaneously to the same file image. This technique is powerful enough to be dangerous, but in the right hands it makes for exceedingly fast implementations.

Character encoding and searching

The final feature of NIO that I want to introduce in this article is charset, a package for converting between different character encodings. Even before NIO, Java built in much of the same functionality through the getBytes method. charset is welcome, though, because it is more flexible than getBytes and easier to implement at a low architectural level, which yields superior performance. This is particularly valuable for searches that must be sensitive to the encoding, collation, and other characteristics of languages other than English.

NETWERKPARAMETERS

Veerle Ongenae



Systems often run with multiple active network connections, such as wired Ethernet, 802.11 b/g (wireless), and bluetooth. Some applications might need to access this information to perform the particular network activity on a specific connection.

The java.net.NetworkInterface class provides access to this information.

This lesson guides you through some of the more common uses of this class and provides examples that list all the network interfaces on a machine as well as their IP addresses and status.

What Is a Network Interface?
This page describes a network interface and explains why you might want to use it.

Retrieving Network Interfaces
This page contains an example that illustrates how a client program can retrieve all the network interfaces on a machine.

Listing Network Interface Addresses
This page shows you how to list the IP addresses assigned to all the network interfaces on a machine.

Network Interface Parameters
This page shows you how to determine whether a network interface is running or if

the network interface is a loopback interface, a point-to-point interface, or a virtual interface. You can also learn how to determine if the interface supports multicasting.

Netwerkinterface

3

```
NetworkInterface nif = NetworkInterface.getByName("wlan2");
Enumeration<InetAddress> nifAddresses = nif.getInetAddresses();

Socket soc = new java.net.Socket();
soc.bind(new InetSocketAddress(nifAddresses.nextElement(), 0));

soc.connect(new InetSocketAddress(address, port));

NetworkInterface nif = NetworkInterface.getByName("eth0");
MulticastSocket ms = new MulticastSocket();
ms.joinGroup(new InetSocketAddress(hostname, port), nif);
```

Industrieel Ingenieur Informatica, UGent

What Is a Network Interface?
A *network interface* is the point of interconnection between a computer and a private or public network. A network interface is generally a **network interface card** (NIC), but does not have to have a **physical** form. Instead, the network interface can be implemented in software. For example, the loopback interface (127.0.0.1 for IPv4 and ::1 for IPv6) is not a physical device but a piece of software **simulating** a network interface. The loopback interface is commonly used in test environments.

The java.net.NetworkInterface class represents both types of interfaces.

NetworkInterface is useful for a multi-homed system, which is a system with multiple NICs. Using NetworkInterface, you can specify which NIC to use for a particular network activity.

For example, assume you have a machine with two configured NICs, and you want to send data to a server. You create a socket like this:
Socket soc = new java.net.Socket();
soc.connect(new InetSocketAddress(address, port));

To send the data, the system determines which interface is used. However, if you have a preference or otherwise need to specify which NIC to use, you can query the system for the appropriate interfaces and find an address on the interface you want to use. When you create the socket and bind it to that address, the system uses the

associated interface. For example:

```
NetworkInterface nif = NetworkInterface.getByName("bge0");
Enumeration<InetAddress> nifAddresses = nif.getInetAddresses();
Socket soc = new java.net.Socket();
soc.bind(new InetSocketAddress(nifAddresses.nextElement(), 0));
soc.connect(new InetSocketAddress(address, port));
```

You can also use NetworkInterface to identify the local interface on which a multicast group is to be joined. For example:

```
NetworkInterface nif = NetworkInterface.getByName("bge0");
MulticastSocket ms = new MulticastSocket();
ms.joinGroup(new InetSocketAddress(hostname, port), nif);
```

NetworkInterface can be used with Java APIs in many other ways beyond the two uses described here.

Ophalen netwerkinterfaces

4

```
Enumeration<NetworkInterface> nets
  = NetworkInterface.getNetworkInterfaces();

  for (NetworkInterface netIf : Collections.list(nets)) {
    out.printf("Display name: %s\n", netIf.getDisplayName());
    out.printf("Name: %s\n", netIf.getName());
    displaySubInterfaces(netIf);
    out.printf("\n");
   }
```

Industrieel Ingenieur Informatica, UGent

The NetworkInterface class has no public constructor. Therefore, you cannot just create a new instance of this class with the new operator. Instead, the following static methods are available so that you can retrieve the interface details from the system:
getByInetAddress(),
getByName(),
and getNetworkInterfaces().

The first two methods are used when you already know the IP address or the name of the particular interface. The third method, getNetworkInterfaces() returns the complete list of interfaces on the machine.

Network interfaces can be hierarchically organized. The NetworkInterface class includes two methods, getParent() and getSubInterfaces(), that are pertinent to a network interface hierarchy. The getParent() method returns the parent NetworkInterface of an interface. If a network interface is a subinterface, getParent() returns a non-null value. The getSubInterfaces() method returns all the subinterfaces of a network interface.

IP-adressen oplijsten

5

```
Enumeration<NetworkInterface> nets
  = NetworkInterface.getNetworkInterfaces();
for (NetworkInterface netint : Collections.list(nets))
  out.printf("Display name: %s\n", netint.getDisplayName());
  out.printf("Name: %s\n", netint.getName());
  Enumeration<InetAddress> inetAddresses
    = netint.getInetAddresses();
  for (InetAddress inetAddress :
          Collections.list(inetAddresses)) {
    out.printf("InetAddress: %s\n", inetAddress);
  }
  out.printf("\n");
```

Industrieel Ingenieur Informatica, UGent

One of the most useful pieces of information you can get from a network interface is the list of IP addresses that are assigned to it. You can obtain this information from a NetworkInterface instance by using one of two methods. The first method, getInetAddresses(), returns an Enumeration of InetAddress. The other method, getInterfaceAddresses(), returns a list of java.net.InterfaceAddress instances. This method is used when you need more information about an interface address beyond its IP address. For example, you might need additional information about the subnet mask and broadcast address when the address is an IPv4 address, and a network prefix length in the case of an IPv6 address.

Parameters netwerkinterface

6

- TestNetwerkParameters.java

Industrieel Ingenieur Informatica, UGent

You can access network parameters about a network interface beyond the name and IP addresses assigned to it.
You can discover if a network interface is “up” (that is, running) with the isUP() method. The following methods indicate the network interface type:

isLoopback() indicates if the network interface is a loopback interface.
isPointToPoint() indicates if the interface is a point-to-point interface.
isVirtual() indicates if the interface is a virtual interface.

The supportsMulticast() method indicates whether the network interface supports multicasting. The getHardwareAddress() method returns the network interface's physical hardware address, usually called MAC address, when it is available. The getMTU() method returns the Maximum Transmission Unit (MTU), which is the largest packet size.

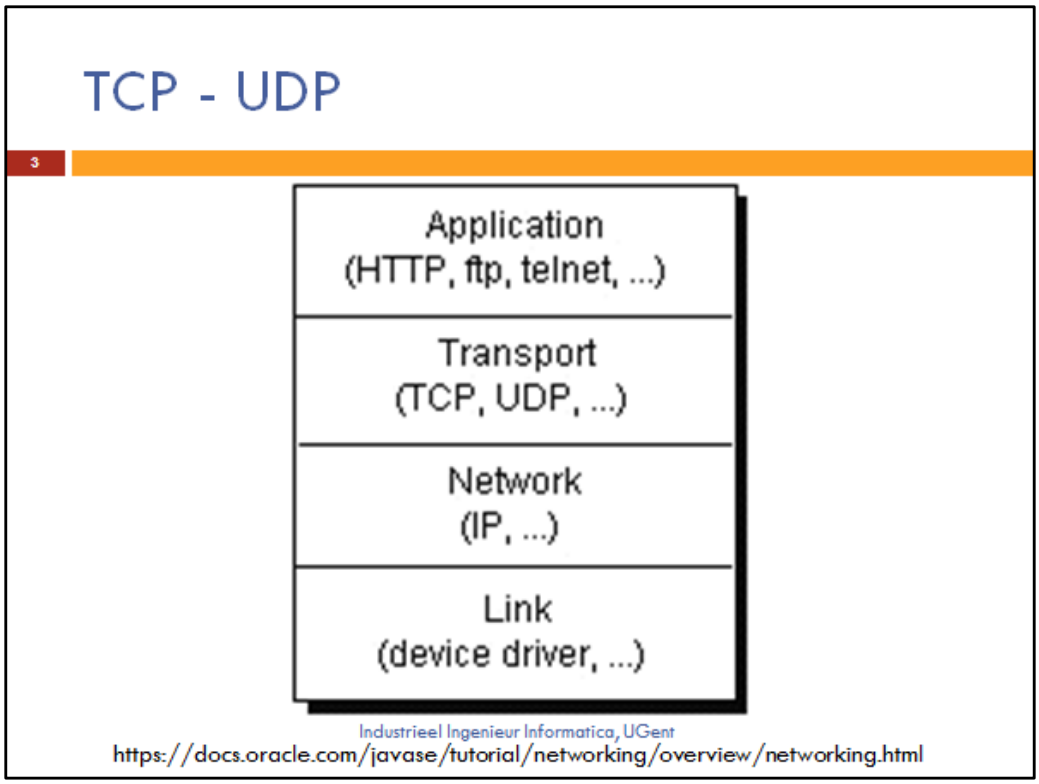
NETWERKPROGRAMMATIE JAVA

Veerle Ongenae

Overzicht

2

- Herhaling



Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

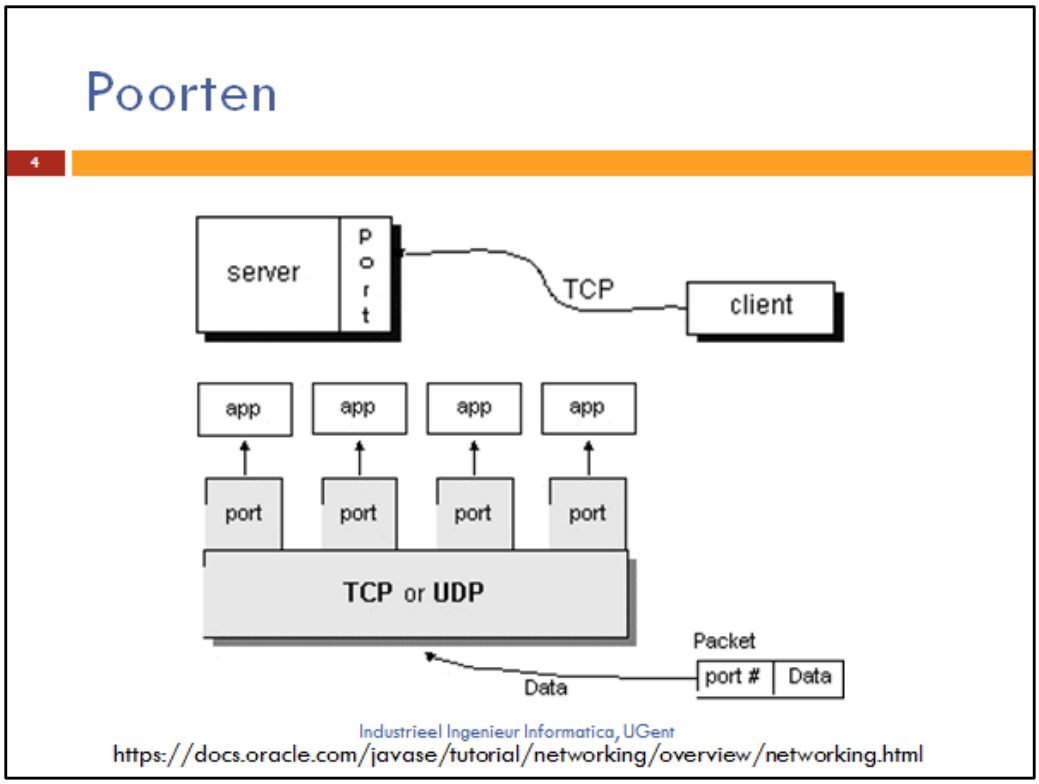
When you write Java programs that communicate over the network, you are programming at the application layer. Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the java.net package. These classes provide system-independent network communication. However, to decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.

TCP (*Transmission Control Protocol*) is a connection-based protocol that provides a reliable flow of data between two computers.
TCP provides a point-to-point channel for applications that require reliable communications. The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, you end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.

UDP (*User Datagram Protocol*) is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival.

UDP is not connection-based like TCP.
The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called *datagrams*, from one application to another. Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is independent of any other.

For many applications, the guarantee of reliability is critical to the success of the transfer of information from one end of the connection to the other. However, other forms of communication don't require such strict standards. In fact, they may be slowed down by the extra overhead or the reliable connection may invalidate the service altogether.
Consider, for example, a clock server that sends the current time to its client when requested to do so. If the client misses a packet, it doesn't really make sense to resend it because the time will be incorrect when the client receives it on the second try. If the client makes two requests and receives packets from the server out of order, it doesn't really matter because the client can figure out that the packets are out of order and make another request. The reliability of TCP is unnecessary in this instance because it causes performance degradation and may hinder the usefulness of the service.
Another example of a service that doesn't need the guarantee of a reliable channel is the ping command. The purpose of the ping command is to test the communication between two programs over the network. In fact, ping needs to know about dropped or out-of-order packets to determine how good or bad the connection is. A reliable channel would invalidate this service altogether.
The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data from one application to another. Sending datagrams is much like sending a letter through the mail service: The order of delivery is not important and is not guaranteed, and each message is independent of any others.



Generally speaking, a computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection. However, the data may be intended for different applications running on the computer. So how does the computer know to which application to forward the data? Through the use of ports.

Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its 32-bit IP address, which IP uses to deliver data to the right computer on the network. Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.

In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port.

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.

Port numbers range from 0 to 65,535 because ports are represented by 16-bit numbers. The port numbers ranging from 0 - 1023 are restricted; they are reserved

for use by well-known services such as HTTP and FTP and other system services. These ports are called *well-known ports*. Your applications should not attempt to bind to them.

Netwerkprogrammatie Java

5

- Package `java.net`
- TCP
  - URL
  - URLConnection
  - Socket
  - ServerSocket
- UDP
  - DatagramPacket
  - DatagramSocket
  - MulticastSocket

Industrieel Ingenieur Informatica, UGent

Networking Classes in the JDK

Through the classes in java.net, Java programs can use TCP or UDP to communicate over the Internet. The URL, URLConnection, Socket, and ServerSocket classes all use TCP to communicate over the network. The DatagramPacket, DatagramSocket, and MulticastSocket classes are for use with UDP.

URL

6

https://docs.oracle.com/javase/tutorial/network/urls/def.html

Industrieel Ingenieur Informatica, UGent

URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet.
You provide URLs to your favorite Web browser so that it can locate files on the Internet in the same way that you provide addresses on letters so that the post office can locate your correspondents.

A URL has two main components:
Protocol identifier: For the URL http://example.com, the protocol identifier is http.
Resource name: For the URL http://example.com, the resource name is example.com.

The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol (FTP), Gopher, File, and News.

The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the following components:
Host Name The name of the machine on which the resource lives.
Filename The pathname to the file on the machine.

Port Number The port number to which to connect (typically optional).
Reference A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).
For many protocols, the host name and the filename are required, while the port number and reference are optional. For example, the resource name for an HTTP URL must specify a server on the network (Host Name) and the path to the document on that machine (Filename); it also can specify a port number and a reference.

Java programs that interact with the Internet also may use URLs to find the resources on the Internet they wish to access. Java programs can use a class called URL in the java.net package to represent a URL address.

Overzicht

7

- Herhaling
- URL's in Java

URL's in Java

8

```
URL myURL =
    new URL("http://example.com/pages/");
URL page1URL = new URL(myURL, "page1.html");
URL page2URL = new URL(myURL, "page2.html");

myURL = new URL("http", "example.com", "/pages/page1.html");

myURL = new URL("http", "example.com",80, "/pages/page1.html");

→ MalformedURLException
```

Industrieel Ingenieur Informatica, UGent

The easiest way to create a URL object is from a String that represents the human-readable form of the URL address. This is typically the form that another person will use for a URL. In your Java program, you can use a String containing this text to create a URL object:

URL myURL = new URL("http://example.com/");

The URL object created above represents an absolute URL. An absolute URL contains all of the information necessary to reach the resource in question. You can also create URL objects from a relative URL address.

Creating a URL Relative to Another
A relative URL contains only enough information to reach the resource relative to (or in the context of) another URL.

In your Java programs, you can create a URL object from a relative URL specification. For example, suppose you know two URLs at the site example.com:
http://example.com/pages/page1.html
http://example.com/pages/page2.html

You can create URL objects for these pages relative to their common base URL: http://example.com/pages/ like this:

```
URL myURL = new URL("http://example.com/pages/");
URL page1URL = new URL(myURL, "page1.html");
URL page2URL = new URL(myURL, "page2.html");
```

This code snippet uses the URL constructor that lets you create a URL object from another URL object (the base) and a relative URL specification. The general form of this constructor is:
URL(URL baseURL, String relativeURL)

The first argument is a URL object that specifies the base of the new URL. The second argument is a String that specifies the rest of the resource name relative to the base. If baseURL is null, then this constructor treats relativeURL like an absolute URL specification. Conversely, if relativeURL is an absolute URL specification, then the constructor ignores baseURL.

This constructor is also useful for creating URL objects for named anchors (also called references) within a file. For example, suppose the page1.html file has a named anchor called BOTTOM at the bottom of the file. You can use the relative URL constructor to create a URL object for it like this:

```
URL page1BottomURL = new URL(page1URL,"#BOTTOM");
```

Other URL Constructors

The URL class provides two additional constructors for creating a URL object. These constructors are useful when you are working with URLs, such as HTTP URLs, that have host name, filename, port number, and reference components in the resource name portion of the URL. These two constructors are useful when you do not have a String containing the complete URL specification, but you do know various components of the URL.
For example, suppose you design a network browsing panel similar to a file browsing panel that allows users to choose the protocol, host name, port number, and filename. You can construct a URL from the panel's components. The first constructor creates a URL object from a protocol, host name, and filename. The following code snippet creates a URL to the page1.html file at the example.com site:

```
new URL("http", "example.com", "/pages/page1.html");
```

This is equivalent to

```
new URL("http://example.com/pages/page1.html");
```

The first argument is the protocol, the second is the host name, and the last is the pathname of the file. Note that the filename contains a forward slash at the beginning. This indicates that the filename is specified from the root of the host.
The final URL constructor adds the port number to the list of arguments used in the previous constructor:

```
URL gamelan = new URL("http", "example.com", 80, "pages/page1.html");
```

This creates a URL object for the following URL:
http://example.com:80/pages/page1.html
If you construct a URL object using one of these constructors, you can get a String containing the complete URL address by using the URL object's toString method or the equivalent toExternalForm method.

URLs are "**write-once**" objects. Once you've created a URL object, you cannot change any of its attributes (protocol, host name, filename, or port number).

Each of the four URL constructors throws a **MalformedURLException** if the arguments to the constructor refer to a null or unknown protocol. Typically, you want to catch and handle this exception by embedding your URL constructor statements in a try/catch pair, like this:

```
try {
   URL myURL = new URL(...);
}
catch (MalformedURLException e) {
   // exception handler code here
   // ...
}
```

URI

9

- http://example.com/hello world/

```
URI uri = new URI("http", "example.com", "/hello world/", "");
URL url = uri.toURL();
```

Industrieel Ingenieur Informatica, UGent

URL addresses with Special characters

Some URL addresses contain special characters, for example the space character. Like this:

http://example.com/hello world/

To make these characters legal they need to be encoded before passing them to the URL constructor.

URL url = new URL("http://example.com/hello%20world");

Encoding the special character(s) in this example is easy as there is only one character that needs encoding, but for URL addresses that have several of these characters or if you are unsure when writing your code what URL addresses you will need to access, you can use the multi-argument constructors of the java.net.URI class to automatically take care of the encoding for you.

URI uri = new URI("http", "example.com", "/hello world/", "");

And then convert the URI to a URL.

URL url = uri.toURL();

Eigenschappen URL

10

```
import java.net.*; import java.io.*;
public class ParseURL {
  public static void main(String[] args) throws Exception {
    URL aURL = new URL("http://example.com:80/docs/books/tutorial"
                + "/index.html?name=networking#DOWNLOADING");
    System.out.println("protocol = " + aURL.getProtocol());
    System.out.println("authority = " + aURL.getAuthority());
    System.out.println("host = " + aURL.getHost());
    System.out.println("port = " + aURL.getPort());
    System.out.println("path = " + aURL.getPath());
    System.out.println("query = " + aURL.getQuery());
    System.out.println("filename = " + aURL.getFile());
    System.out.println("ref = " + aURL.getRef());
  }
```

```
protocol = http
authority = example.com:80
host = example.com
port = 80
path = /docs/books/tutorial/index.htm
query = name=networking
```

```
filename = /docs/books/tutorial/index.html?name=networking
ref = DOWNLOADING
```

The URL class provides several methods that let you query URL objects. You can get the protocol, authority, host name, port number, path, query, filename, and reference from a URL using these accessor methods:
getProtocol Returns the protocol identifier component of the URL.
getAuthority Returns the authority component of the URL.
getHost Returns the host name component of the URL.
getPort Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.
getPath Returns the path component of this URL.
getQuery Returns the query component of this URL.
getFile Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.
getRef Returns the reference component of the URL.

Remember that not all URL addresses contain these components. The URL class provides these methods because HTTP URLs do contain these components and are perhaps the most commonly used URLs. The URL class is somewhat HTTP-centric. You can use these get*XXX* methods to get information about the URL regardless of the constructor that you used to create the URL object.
The URL class, along with these accessor methods, frees you from ever having to parse URLs again! Given any string specification of a URL, just create a new URL object and call any of the accessor methods for the information you need.

Lezen van een URL

11

```
try {
  URL ugent = new URL("http://www.ugent.be/");
  try (BufferedReader in = new BufferedReader(
    new InputStreamReader(ugent.openStream()))) {
      String inputLine;
      while ((inputLine = in.readLine()) != null) {
        System.out.println(inputLine);
      }
  } catch (IOException ex) {
    Logger.getLogger(
      VBUrls.class.getName()).log(Level.SEVERE, null, ex);
  }
} catch (MalformedURLException ex) {
  Logger.getLogger(
    VBUrls.class.getName()).log(Level.SEVERE, null, ex);
}
```

Industrieel Ingenieur Informatica, UGent

After you've successfully created a URL, you can call the URL's openStream() method to get a stream from which you can read the contents of the URL. The openStream() method returns a java.io.InputStream object, so reading from a URL is as easy as reading from an input stream.

The following small Java program uses openStream() to get an input stream on the URL http://www.ugent.be/. It then opens a BufferedReader on the input stream and reads from the BufferedReader thereby reading from the URL. Everything read is copied to the standard output stream.

Communicatie met URL

12

```
try {
  URL url = new URL("http://localhost:8080/VbServlets/hallo2");
  URLConnection connection = url.openConnection();
  connection.setDoOutput(true);
  try (OutputStreamWriter out = new OutputStreamWriter(connection.getOutputStrea
    String naam = URLEncoder.encode("CD&V", "UTF-8");
    out.write("naam=" + naam);
  }
  BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInpu
  String line;
  while ((line = in.readLine()) != null) {
    System.out.println(line);
  }
  in.close();
} catch (IOException ex) {
  Logger.getLogger(VBUrls.class.getName()).log(Level.SEVERE, null, ex);
}
```

Industrieel Ingenieur Informatica, UGent

The URLConnection class contains many methods that let you communicate with the URL over the network. URLConnection is an HTTP-centric class; that is, many of its methods are useful only when you are working with HTTP URLs. However, most URL protocols allow you to read from and write to the connection.

Writing to a URLConnection

Many HTML pages contain *forms* — text fields and other GUI objects that let you enter data to send to the server. After you type in the required information and initiate the query by clicking a button, your Web browser writes the data to the URL over the network. At the other end the server receives the data, processes it, and then sends you a response, usually in the form of a new HTML page.

Many of these HTML forms use the HTTP POST METHOD to send data to the server. Thus writing to a URL is often called *posting to a URL*. The server recognizes the POST request and reads the data sent from the client.

For a Java program to interact with a server-side process it simply must be able to write to a URL, thus providing data to the server. It can do this by following these steps:

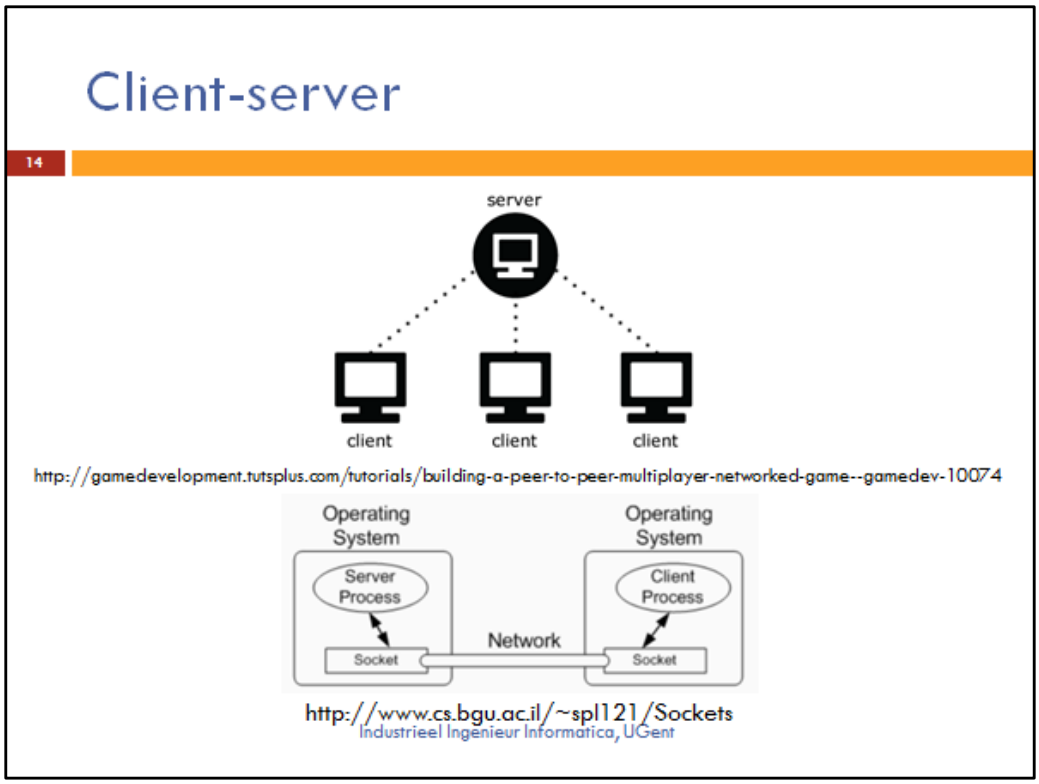
Create a URL.
Retrieve the URLConnection object.
Set output capability on the URLConnection.
Open a connection to the resource.

Get an output stream from the connection.
Write to the output stream.
Close the output stream.

Overzicht

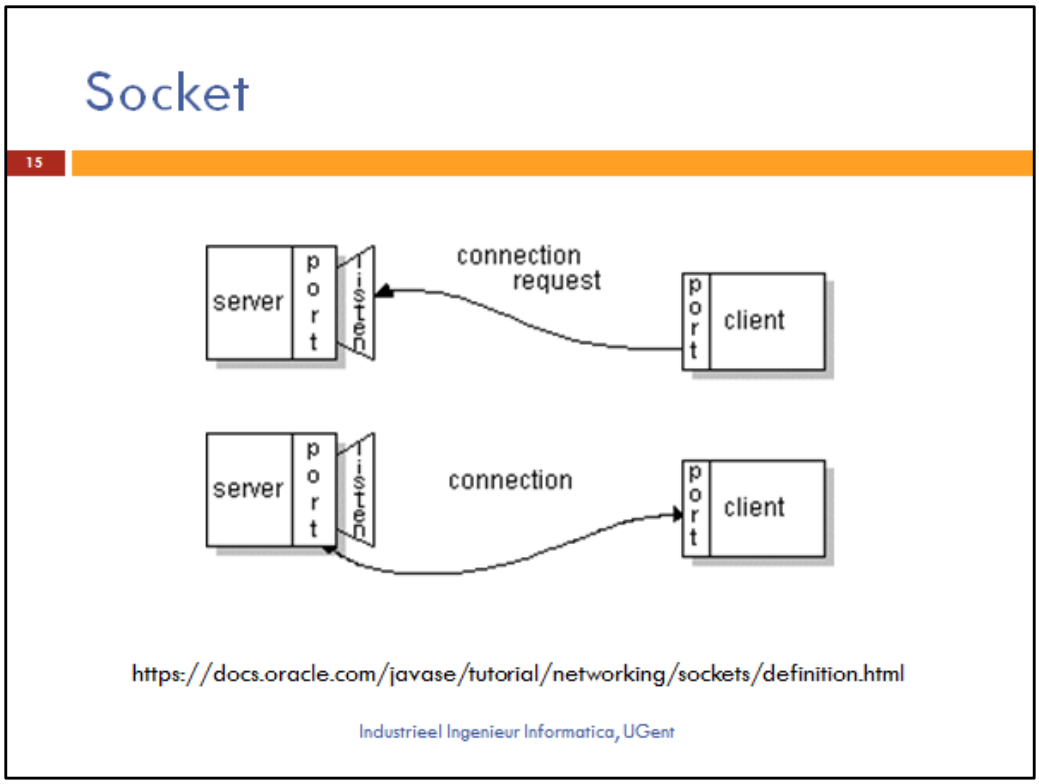
13

- Herhaling
- URL's in Java
- Client-Serverapplicatie



URLs and URLConnections provide a relatively high-level mechanism for accessing resources on the Internet. Sometimes your programs require lower-level network communication, for example, when you want to write a client-server application. In **client-server applications**, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it. TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

A **socket** is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes--Socket and ServerSocket--that implement the client side of the connection and the server side of the connection, respectively.



Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.
On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

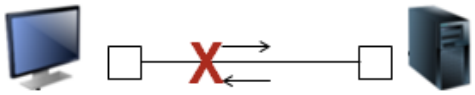
On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.
The client and server can now communicate by writing to or reading from their sockets.

A *socket* is one **endpoint** of a **two-way communication** link between two programs

running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.
An endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

The java.net package in the Java platform provides a class, **Socket**, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the java.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion. Additionally, java.net includes the **ServerSocket** class, which implements a socket that servers can use to listen for and accept connections to clients.

Client



16

```
String hostName = "localhost";
int portNumber = 4444;

try (Socket echoSocket = new Socket(hostName, portNumber);
     PrintWriter out = new PrintWriter(echoSocket.getOutputStream(), true);
     BufferedReader in = new BufferedReader(
       new InputStreamReader(echoSocket.getInputStream()));
     BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in))) {

  String userInput = leesTekst(stdIn);
  while (!userInput.equals("exit")) {
    out.println(userInput); // naar server
    System.out.println("echo: " + in.readLine()); // van server
    userInput = leesTekst(stdIn);
  }
} catch (IOException ex) {
    Logger.getLogger(EchoClient.class.getName()).log(Level.SEVERE, null, ex);
}
```

Industrieel Ingenieur Informatica, UGent

Let's look at a simple example that illustrates how a program can establish a connection to a server program using the Socket class and then, how the client can send data to and receive data from the server through the socket.

The example program implements a client, EchoClient, that connects to an echo server. The echo server receives data from its client and echoes it back.
The EchoClient example creates a socket, thereby getting a connection to the echo server. It reads input from the user on the standard input stream, and then forwards that text to the echo server by writing the text to the socket. The server echoes the input back through the socket to the client. The client program reads and displays the data passed back to it from the server.
Note that the EchoClient example both writes to and reads from its socket, thereby sending data to and receiving data from the echo server.

The statements in the try-with-resources statement in the EchoClient example are critical. These lines establish the socket connection between the client and the server and open a PrintWriter and a BufferedReader on the socket.
The first statement in the try-with resources statement creates a new Socket object and names it echoSocket. The Socket constructor used here requires the name of the computer and the port number to which you want to connect. When you run this program on your computer, make sure that the host name you use is the fully qualified IP name of the computer to which you want to connect.

The second statement in the try-with resources statement gets the socket's output stream and opens a PrintWriter on it. Similarly, the third statement gets the socket's input stream and opens a BufferedReader on it. The example uses readers and writers so that it can write Unicode characters over the socket.
To send data through the socket to the server, the EchoClient example needs to write to the PrintWriter. To get the server's response, EchoClient reads from the BufferedReader object stdIn, which is created in the fourth statement in the try-with resources statement.

The next interesting part of the program is the while loop. The loop reads a line at a time from the standard input stream and immediately sends it to the server by writing it to the PrintWriter connected to the socket.
The second statement in the while loop reads a line of information from the BufferedReader connected to the socket. The readLine method waits until the server echoes the information back to EchoClient. When readline returns, EchoClient prints the information to the standard output.

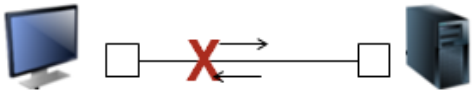
The while loop continues until the user types “exit”. That is, the EchoClient example reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until the user types “exit”. The while loop then terminates, and the Java runtime automatically closes the readers and writers connected to the socket and to the standard input stream, and it closes the socket connection to the server. The Java runtime closes these resources automatically because they were created in the try-with-resources statement. The Java runtime closes these resources in reverse order that they were created. (This is good because streams connected to a socket should be closed before the socket itself is closed.)

This client program is straightforward and simple because the echo server implements a simple protocol. The client sends text to the server, and the server echoes it back. When your client programs are talking to a more complicated server such as an HTTP server, your client program will also be more complicated. However, the basics are much the same as they are in this program:
Open a socket.
Open an input stream and output stream to the socket.
Read from and write to the stream according to the server's protocol.
Close the streams.
Close the socket.

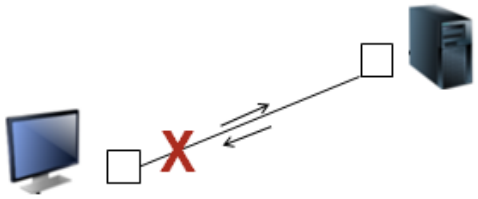
Only step 3 differs from client to client, depending on the server. The other steps remain largely the same.

Server

17



Server



KnockKnockServer

19

```
← Knock! Knock!
→ Who's there?
← Turnip
→ Turnip who?
← Turnip the heat, it's cold in here! Want another? (y/n)
→ y
← Knock! Knock!
→ Who's there?
← Little Old Lady
→ Little Old Lady who?
← I didn't know you could yodel! Want another? (y/n)
→ y
← Knock! Knock!
→ Who's there?
← Atch
→ Atch who?
← Bless you! Want another? (y/n)
→ n
← Bye.
```

KnockKnockServer

20

```
try {
  ServerSocket serverSocket = new ServerSocket(4444);
  while (true) {
    try (Socket clientSocket = serverSocket.accept())
      try (PrintWriter out
           = new PrintWriter(clientSocket.getOutputStream(), true);
          BufferedReader in = new BufferedReader(
           new InputStreamReader(clientSocket.getInputStream()))){
        String inputLine, outputLine;
        while ((inputLine = in.readLine()) != null) {
          outputLine = kkp.processInput(inputLine);
          out.println(outputLine);
          if (outputLine.equals("Bye.")) {break;}
        }
      } …
```

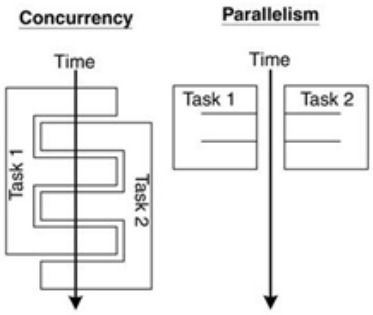
Overzicht

21

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency

Concurrency

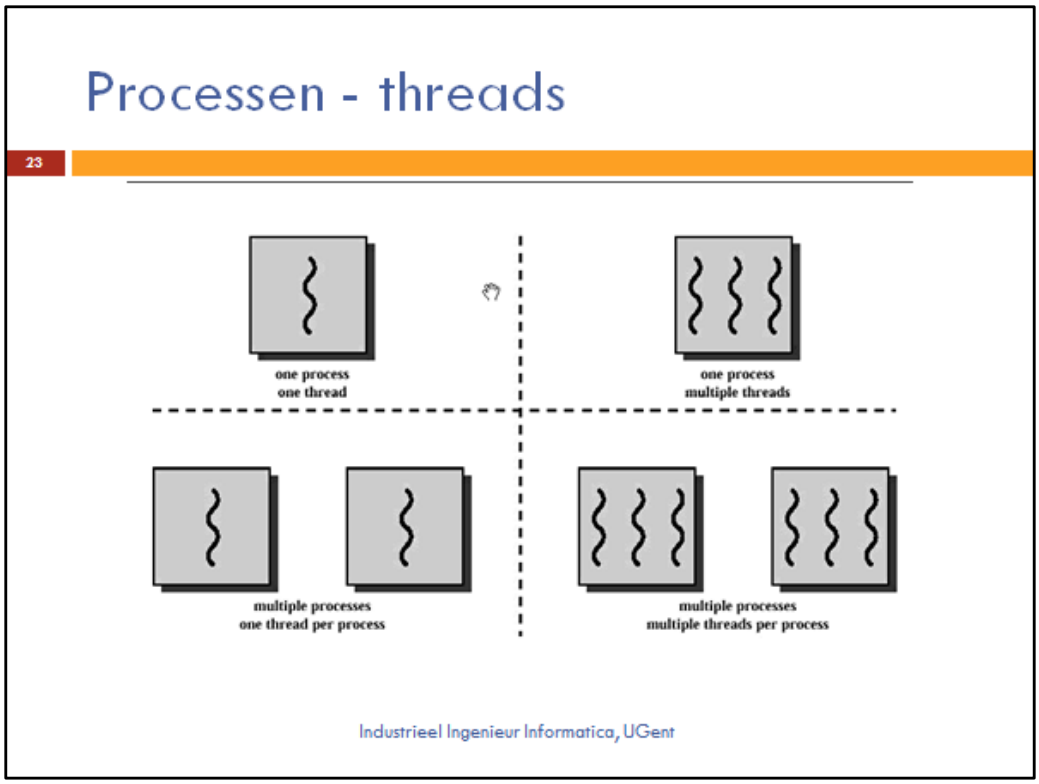
22



http://sourcecodemania.com/thread-programming-in-java/

Industrieel Ingenieur Informatica, UGent

Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio. Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display. Software that can do such things is known as *concurrent* software.
The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs. This lesson introduces the platform's basic concurrency support and summarizes some of the high-level APIs in the java.util.concurrent packages.



In concurrent programming, there are two basic units of execution: processes and threads. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores.

Processes
A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating

processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a ProcessBuilder object.

Threads

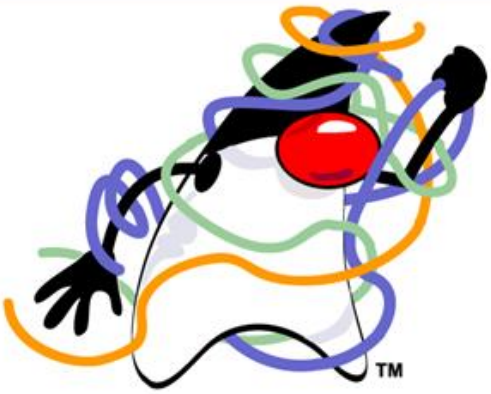
Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads.

Thread - Executor

24



http://crunchify.com/java-simple-thread-example/

Industrieel Ingenieur Informatica, UGent

Each thread is associated with an instance of the class Thread. There are two basic strategies for using Thread objects to create a concurrent application.

To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.

To abstract thread management from the rest of your application, pass the application's tasks to an executor.

Overzicht

25

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads

Thread – optie 1

26

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

Industrieel Ingenieur Informatica, UGent

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

Provide a Runnable object. The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor.

Thread – optie 2

27

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Industrieel Ingenieur Informatica, UGent

Subclass Thread. The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run.

Notice that both examples invoke Thread.start in order to start the new thread.

Which of these idioms should you use? The first idiom, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread. The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread.

The Thread class defines a number of methods useful for thread management. These include static methods, which provide information about, or affect the status of, the thread invoking the method. The other methods are invoked from other threads involved in managing the thread and Thread object: sleep, interrupt, join, …

Overzicht

28

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads
  - Synchronisatie

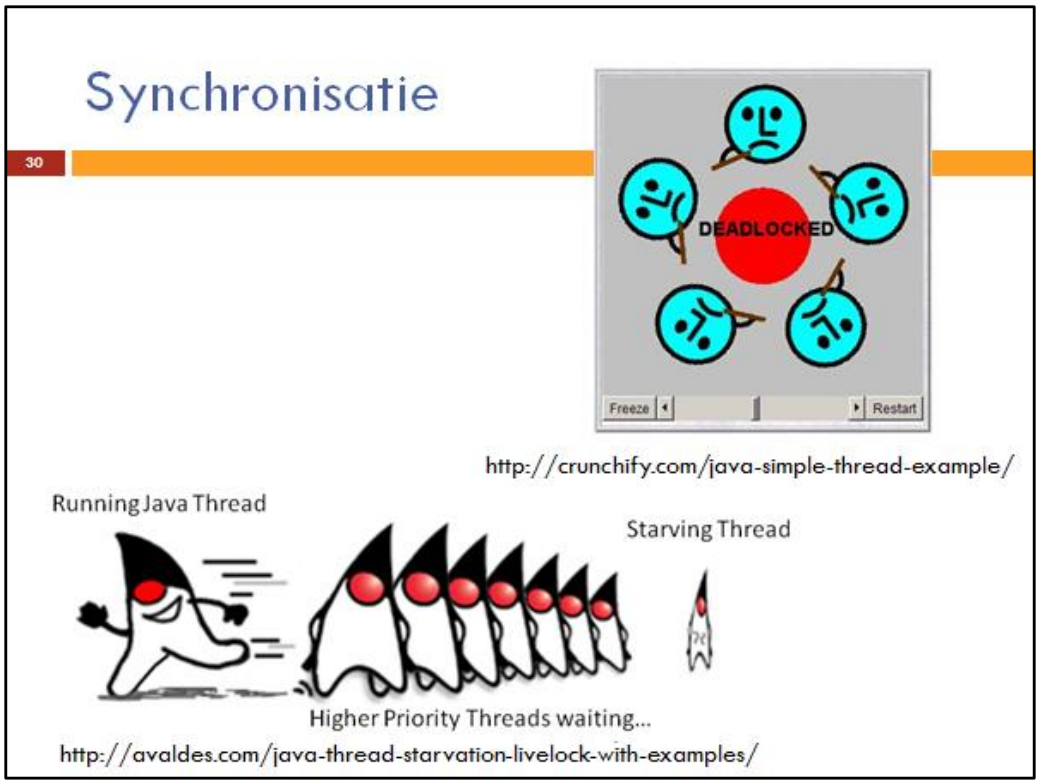
Data delen

29

Industrieel Ingenieur Informatica, UGent

http://blog.jbaysolutions.com/2011/10/06/java-threading-and-concurrency-introduction/

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors. The tool needed to prevent these errors is synchronization.



However, synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Deadlock, starvation and livelock are forms of thread contention.

Thread Interference: errors are introduced when multiple threads access shared data.

Memory Consistency Errors: errors that result from inconsistent views of shared memory.

Synchronized Methods: a simple idiom that can effectively prevent thread interference and memory consistency errors.

Implicit Locks and Synchronization: a more general synchronization idiom, synchronization is based on implicit locks.

Atomic Access: operations that can't be interfered with by other threads, volatile instance variables.

High level concurrency

31

- Lock-objecten
- Executors
- Concurrent collections
- Atomic variables
- ThreadLocalRandom

Industrieel Ingenieur Informatica, UGent

So far, this lesson has focused on the low-level APIs that have been part of the Java platform from the very beginning. These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks. This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

In this section we'll look at some of the high-level concurrency features introduced with version 5.0 of the Java platform. Most of these features are implemented in the new java.util.concurrent packages. There are also new concurrent data structures in the Java Collections Framework.

Lock objects support locking idioms that simplify many concurrent applications.

Executors define a high-level API for launching and managing threads. Executor implementations provided by java.util.concurrent provide thread pool management suitable for large-scale applications.

Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.

Atomic variables have features that minimize synchronization and help avoid memory consistency errors.

ThreadLocalRandom (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.

Overzicht

32

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads
  - Synchronisatie
  - Executor

Executor

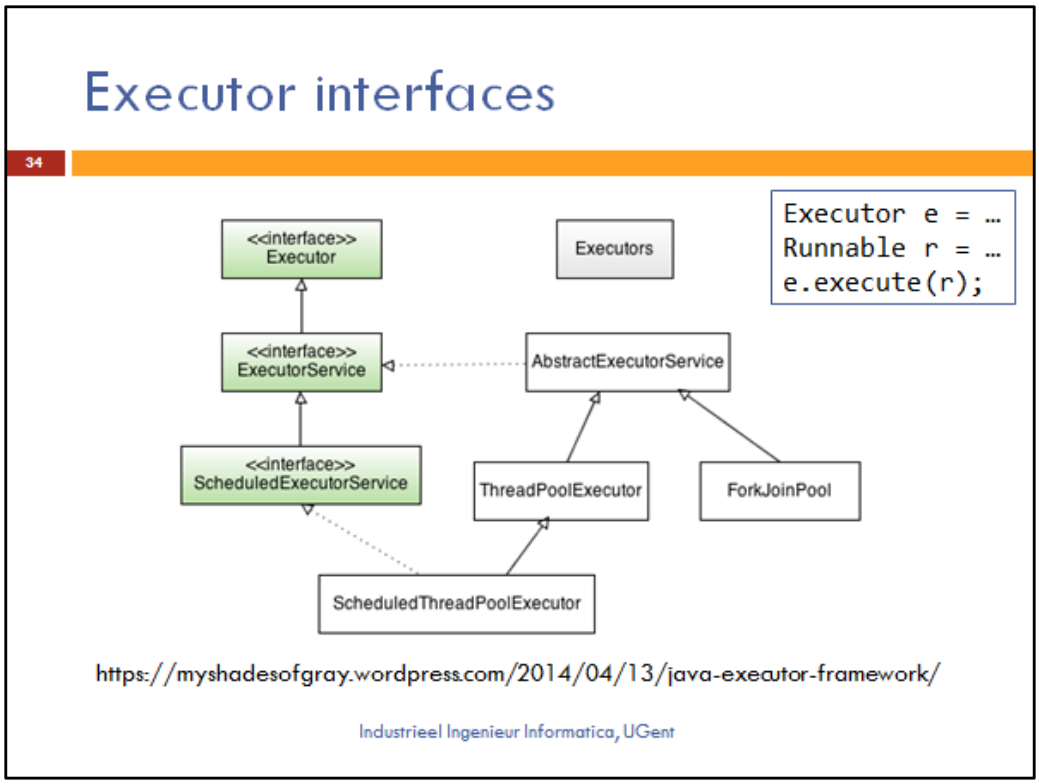
33

- Scheiding
  - Beheren threads
  - Aanmaken threads

Industrieel Ingenieur Informatica, UGent

In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its Runnable object, and the thread itself, as defined by a Thread object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as executors. The following subsections describe executors in detail.

Executor Interfaces define the three executor object types.
Thread Pools are the most common kind of executor implementation.
Fork/Join is a framework (new in JDK 7) for taking advantage of multiple processors.



The java.util.concurrent package defines three executor interfaces:

Executor, a simple interface that supports launching new tasks.
ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

The Executor Interface
The Executor interface provides a single method, execute, designed to be a drop-in replacement for a common thread-creation idiom. If r is a Runnable object, and e is an Executor object you can replace

```
(new Thread(r)).start();
```

with

```
e.execute(r);
```

However, the definition of execute is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the Executor implementation, execute may do the same thing, but is more likely to use an existing worker thread to run r, or to place r in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on Thread Pools.)

The executor implementations in java.util.concurrent are designed to make full use of the more advanced ExecutorService and ScheduledExecutorService interfaces, although they also work with the base Executor interface.

The ExecutorService Interface

The ExecutorService interface supplements execute with a similar, but more versatile submit method. Like execute, submit accepts Runnable objects, but also accepts Callable objects, which allow the task to return a value. The submit method returns a Future object, which is used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks.

ExecutorService also provides methods for submitting large collections of Callable objects. Finally, ExecutorService provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

The ScheduledExecutorService Interface

The ScheduledExecutorService interface supplements the methods of its parent ExecutorService with schedule, which executes a Runnable or Callable task after a specified delay. In addition, the interface defines scheduleAtFixedRate and scheduleWithFixedDelay, which executes specified tasks repeatedly, at defined intervals.

ExecutorService - aanmaken

35

```
ExecutorService execServ1 = Executors.newSingleThreadExecutor();

ExecutorService execServ2 = Executors.newFixedThreadPool(10);

ExecutorService execServ3 = Executors.newScheduledThreadPool(10);
```

Industrieel Ingenieur Informatica, UGent

How you create an ExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ExecutorService instances too.

ExecutorService - gebruik

36

```
executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});
```

```
executorService.submit(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});
```

```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

System.out.println("future.get() = " + future.get());
```

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Future object can be used to check if the Runnable as finished executing.

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.
The Callable's result can be obtained via the Future object returned by the submit(Callable) method.

ExecutorService – reeks opdrachten

37

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(…);
callables.add(…);

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){
    System.out.println("future.get = " + future.get());
}

executorService.shutdown();
```

Industrieel Ingenieur Informatica, UGent

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.
Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a Future to tell the difference.

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.
For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.
To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorService will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down. All tasks submitted to the ExecutorService before shutdown() is called, are executed.
If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

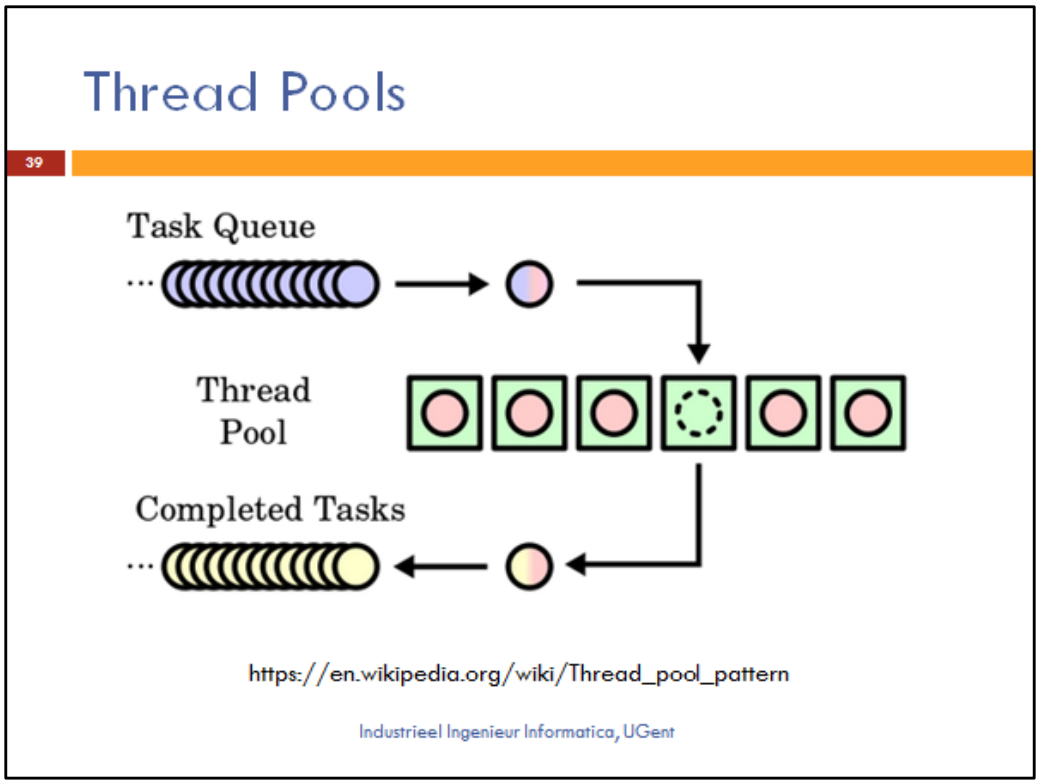
ExecutorService – reeks opdrachten

38

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(…);
callables.add(…);

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();
```

Industrieel Ingenieur Informatica, UGent

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.
If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.



Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it degrade gracefully. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be

servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

A simple way to create an executor that uses a fixed thread pool is to invoke the newFixedThreadPool factory method in java.util.concurrent.Executors This class also provides the following factory methods:

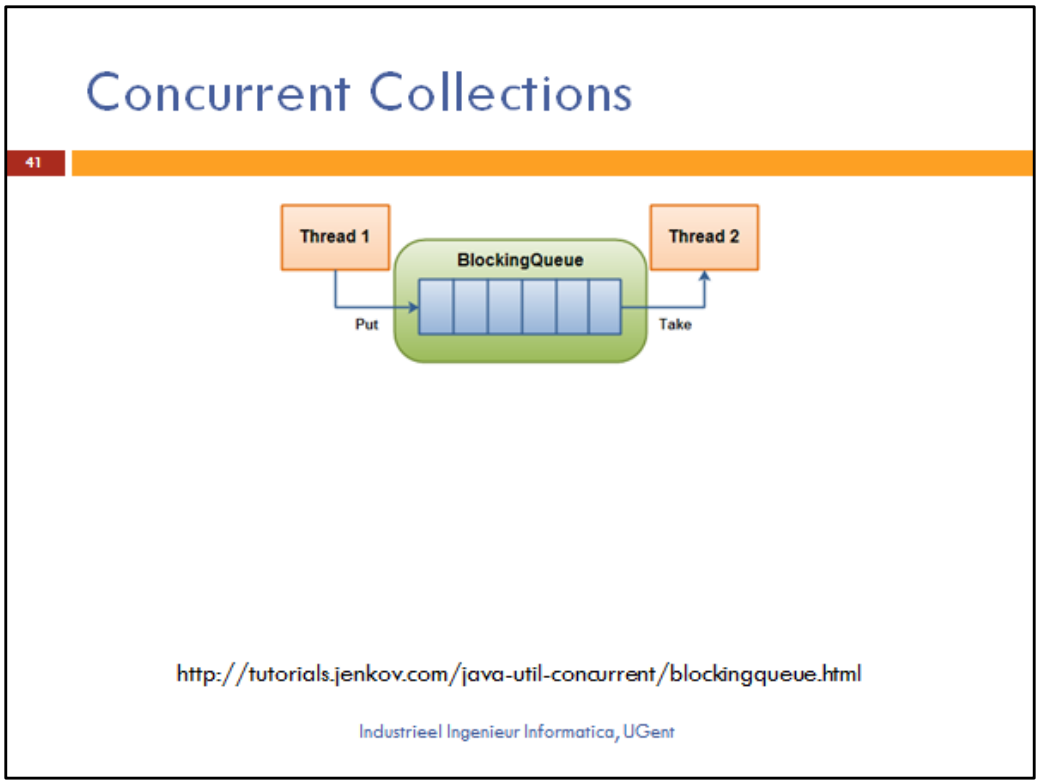
The newCachedThreadPool method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
The newSingleThreadExecutor method creates an executor that executes a single task at a time.
Several factory methods are ScheduledExecutorService versions of the above executors.

If none of the executors provided by the above factory methods meet your needs, constructing instances of java.util.concurrent.ThreadPoolExecutor or java.util.concurrent.ScheduledThreadPoolExecutor will give you additional options.

Overzicht

40

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads
  - Synchronisatie
  - Executor
  - Concurrent Collections



The java.util.concurrent package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

BlockingQueue defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.

ConcurrentMap is a subinterface of java.util.Map that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of ConcurrentMap is ConcurrentHashMap, which is a concurrent analog of HashMap.

ConcurrentNavigableMap is a subinterface of ConcurrentMap that supports approximate matches. The standard general-purpose implementation of ConcurrentNavigableMap is ConcurrentSkipListMap, which is a concurrent analog of TreeMap.

All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

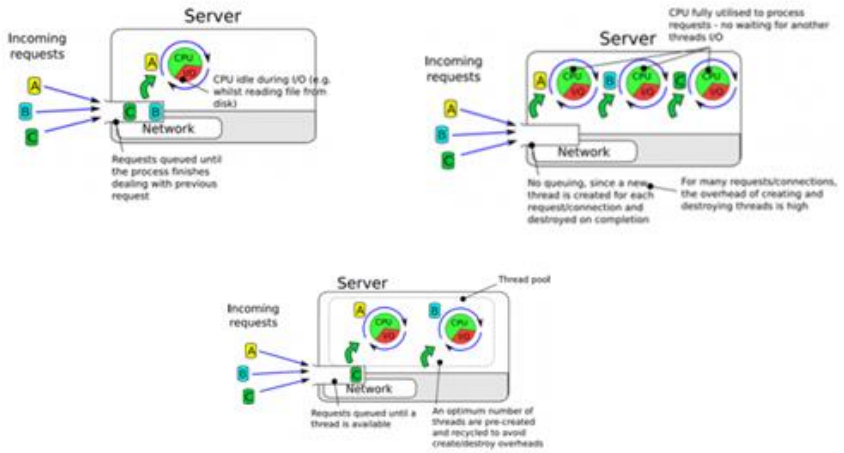
Overzicht

42

- Herhaling
- URL's in Java
- Client-Serverapplicatie
- Concurrency
  - Threads
  - Synchronisatie
  - Executor
  - Concurrent Collections
- Multithreaded Server

Singlethreaded Server

43



https://myshadesofgray.wordpress.com/2014/04/13/java-executor-framework/

KnockKnockServer

44

```
int portNumber = 9999;
boolean listening = true;
try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
  while (listening) {
    new KnockKnockThread(serverSocket.accept()).start();
  }
} catch (IOException e) {
  Logger.getLogger(
    KnockKnockMultiServer.class.getName()).log(Level.SEVERE,
       null, e);
  System.err.println("Could not listen on port " + portNumber);
  throw new RuntimeException(e);
}
```

Industrieel Ingenieur Informatica, UGent

This example describes a simple multithreaded server implemented in Java. The code is based on the singlethreaded KnockKnockServer. The main difference is the server loop. Rather than processing the incoming requests in the same thread that accepts the client connection, the connection is handed off to a worker thread that will process the request.

Note: This code uses a "thread per connection" design which most of us originally thought less efficient than a thread pooled server.

Rather than processing the incoming requests in the same thread that accepts the client connection, the connection is handed off to a worker thread that processes the request. That way the thread listening for incoming requests spends as much time as possible in the serverSocket.accept() call. That way the risk is minimized for clients being denied access to the server because the listening thread is not inside the accept() call.

Multithreaded Server Advantages
The advantages of a multithreaded server compared to a singlethreaded server are summed up below:

Less time is spent outside the accept() call.
Long running client requests do not block the whole server

As mentioned earlier the more time the thread calling serverSocket.accept() spends inside this method call, the more responsive the server will be. Only when the listening thread is inside the accept() call can clients connect to the server. Otherwise the clients just get an error or have to wait.

In a singlethreaded server long running requests may make the server unresponsive for a long period. This is not true for a multithreaded server, unless the long-running request takes up all CPU time time and/or network bandwidth.

KnockKnockServer - Thread

45

```
public class KnockKnockThread extends Thread {
  private Socket socket = null;
  public KnockKnockThread(Socket socket) {
    super("KKMultiServerThread");
    this.socket = socket;
  }
  @Override
  public void run() {
    try (
      PrintWriter out = new PrintWriter(socket.getOutputStream(), true),
      BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));)
      {…}
      socket.close();
```

KnockKnockServer - Executor

46

```
int portNumber = 9999;
boolean listening = true;
try (ServerSocket serverSocket = new ServerSocket(portNumber)) {
  ExecutorService execServ = Executors.newFixedThreadPool(10);
  while (listening) {
    try {
      Socket clientSocket = serverSocket.accept();
      execServ.submit(new KnockKnockRunnable(clientSocket));
    } catch (IOException ex) {… }
  }
} catch (IOException e) {
  Logger.getLogger(
    KnockKnockPool.class.getName()).log(Level.SEVERE, null, e);
  System.err.println("Could not listen on port " + portNumber);
  throw new RuntimeException(e);
}
```

Industrieel Ingenieur Informatica, UGent

This example describes a simple thread pooled server implemented in Java. The code is based on the multithreaded KnockKnock server. The main difference is the server loop. Rather than starting a new thread per incoming connection, the connection is wrapped in a Runnable and handed off to a thread pool with a fixed number of threads. The Runnable's are kept in a queue in the thread pool. When a thread in the thread pool is idle it will take a Runnable from the queue and execute it.

Rather than starting a new thread per incoming connection, the KnockKnockRunnable is passed to the thread pool for execution when a thread in the pool becomes idle.

Thread Pooled Server Advantages

The advantages of a thread pooled server compared to a multithreaded server is that you can control the maximum number of threads running at the same time. This has certain advantages.

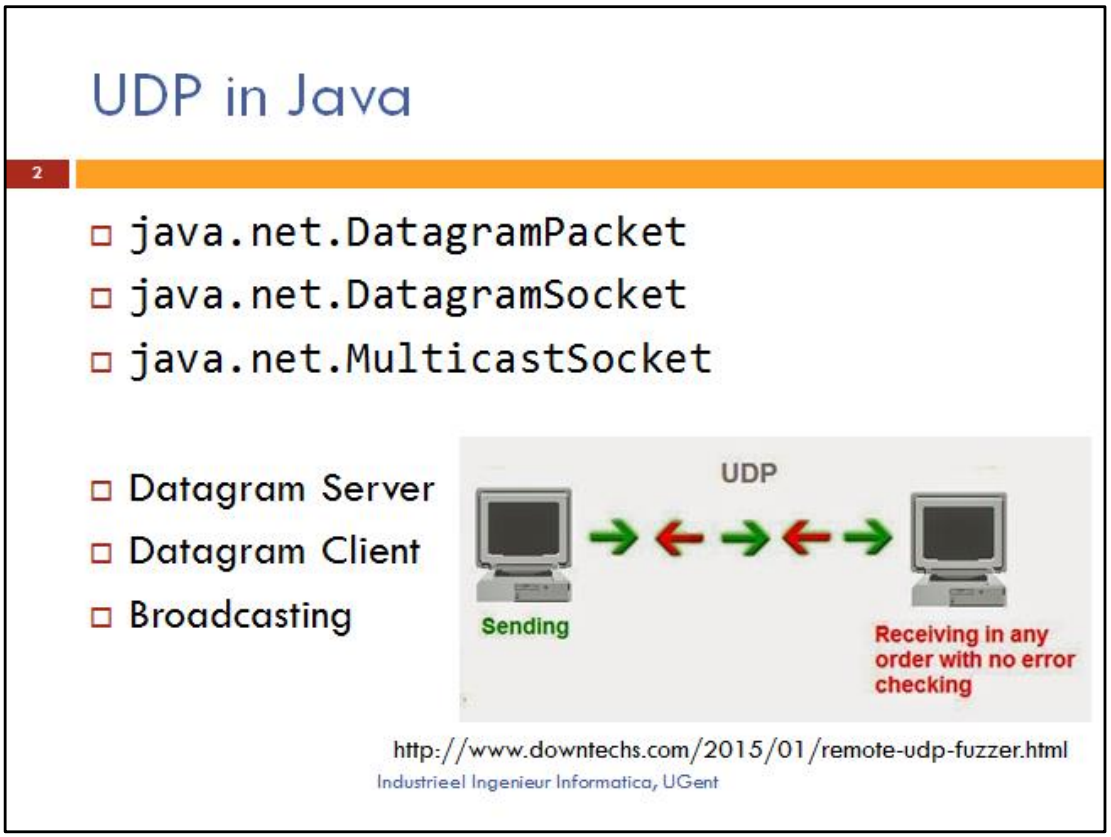
First of all if the requests require a lot of CPU time, RAM or network bandwidth, this may slow down the server if many requests are processed at the same time. For instance, if memory consumption causes the server to swap memory in and out of disk, this will result in a serious performance penalty. By controlling the maximum number of threads you can minimize the risk of resource depletion, both due to limiting the memory taken by the processing of the requests, but also due to the limitation and reuse of the threads. Each thread take up a certain amount of memory too, just to represent the thread itself.

Additionally, executing many requests concurrently will slow down all requests processed. For instance, if you process 1.000 requests concurrently and each request takes 1 second, then all requests will take 1.000 seconds to complete. If you instead queue the requests up and process them say 10 at a time, the first 10 requests will complete after 10 seconds, the next 10 will complete after 20 seconds etc. Only the last 10 requests will complete after 1.000 seconds. This gives a better service to the clients.

UDP

Veerle Ongenae

Some applications that you write to communicate over the network will not require the reliable, point-to-point channel provided by TCP. Rather, your applications might benefit from a mode of communication that delivers independent packages of information whose arrival and order of arrival are not guaranteed.



The UDP protocol provides a mode of network communication whereby applications send packets of data, called datagrams, to one another. A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed. The DatagramPacket and DatagramSocket classes in the java.net package implement system-independent datagram communication using UDP.

Clients and servers that communicate via a reliable channel, such as a TCP socket, have a dedicated point-to-point channel between themselves, or at least the illusion of one. To communicate, they establish a connection, transmit the data, and then close the connection. All data sent over the channel is received in the same order in which it was sent. This is guaranteed by the channel.

In contrast, applications that communicate via datagrams send and receive completely independent packets of information. These clients and servers do not have and do not need a dedicated point-to-point channel. The delivery of datagrams to their destinations is not guaranteed. Nor is the order of their arrival.

[What Is a Datagram?]

Definition: A *datagram* is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

The java.net package contains three classes to help you write Java programs that use datagrams to send and receive packets over the network: DatagramSocket, DatagramPacket, and MulticastSocket. An application can send and receive DatagramPackets through a DatagramSocket. In addition, DatagramPackets can be broadcast to multiple recipients all listening to a MulticastSocket.

Datagram Server

3

- Maak `DatagramSocket`
- Lus
  - Maak leeg `DatagramPacket`
  - Wacht op pakket
  - Bepaal poort en adres van de client
  - Maak `DatagramPacket` met antwoord
  - Verstuur antwoord
- Sluit socket

Datagram Server

4

```
DatagramSocket socket = new DatagramSocket(4445);
boolean moreQuotes = true;
while (moreQuotes) {
  byte[] buf = new byte[256];
  DatagramPacket packet = new DatagramPacket(buf, buf.length);
  socket.receive(packet); // receive request

  String dString = …; // figure out response
  buf = dString.getBytes();

  InetAddress address = packet.getAddress();
  int port = packet.getPort();
  packet = new DatagramPacket(buf, buf.length, address, port);
  socket.send(packet); // send the response to the client at "address" and "port"
}
socket.close();
```

Industrieel Ingenieur Informatica, UGent

The server continuously receives datagram packets over a datagram socket. Each datagram packet received by the server indicates a client request for a quotation. When the server receives a datagram, it replies by sending a datagram packet that contains a one-line "quote of the moment" back to the client.

When created, the server creates a DatagramSocket on port 4445 (arbitrarily chosen). This is the DatagramSocket through which the server communicates with all of its clients.
Remember that certain ports are dedicated to well-known services and you cannot use them. If you specify a port that is in use, the creation of the DatagramSocket will fail.
The constructor also opens a BufferedReader on a file named one-liners.txt which contains a list of quotes. Each quote in the file is on a line by itself.

The run method contains a while loop that continues as long as there are more quotes in the file. During each iteration of the loop, the thread waits for a DatagramPacket to arrive over the DatagramSocket. The packet indicates a request from a client. In response to the client's request, the QuoteServerThread gets a quote from the file, puts it in a DatagramPacket and sends it over the DatagramSocket to the client that asked for it.

Let's look first at the section that receives the requests from clients:

```
byte[] buf = new byte[256];
```

```
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
```

The first statement creates an array of bytes which is then used to create a **DatagramPacket**. The DatagramPacket will be used to receive a datagram from the socket because of the constructor used to create it. This constructor requires only two arguments: a byte array that contains client-specific data and the length of the byte array. When constructing a DatagramPacket to send over the DatagramSocket, you also must supply the Internet address and port number of the packet's destination. You'll see this later when we discuss how the server responds to a client request.

The last statement in the previous code snippet receives a datagram from the socket (the information received from the client gets copied into the packet). The **receive** method waits forever until a packet is received. If no packet is received, the server makes no further progress and just waits.
Now assume that, the server has received a request from a client for a quote. Now the server must respond. This section of code in the run method constructs the response:

```
String dString = null;
if (in == null) dString = new Date().toString();
else dString = getNextQuote();
buf = dString.getBytes();
```

If the quote file did not get opened for some reason, then in equals null. If this is the case, the quote server serves up the time of day instead. Otherwise, the quote server gets the next quote from the already opened file. Finally, the code converts the string to an array of bytes.

Now, the run method sends the response to the client over the DatagramSocket with this code:

```
InetAddress address = packet.getAddress();
int port = packet.getPort();
packet = new DatagramPacket(buf, buf.length, address, port);
socket.send(packet);
```

The first two statements in this code segment get the Internet address and the port number, respectively, from the datagram packet received from the client. The Internet address and port number indicate where the datagram packet came from. This is where the server must send its response. In this example, the byte array of the datagram packet contains no relevant information. The arrival of the packet itself indicates a request from a client that can be found at the Internet address and port number indicated in the datagram packet.

The third statement creates a new DatagramPacket object intended for sending a

datagram message over the datagram socket. You can tell that the new DatagramPacket is intended to send data over the socket because of the constructor used to create it. This constructor requires four arguments. The first two arguments are the same required by the constructor used to create receiving datagrams: a byte array containing the message from the sender to the receiver and the length of this array. The next two arguments are different: an Internet address and a port number. These two arguments are the complete address of the destination of the datagram packet and must be supplied by the sender of the datagram. The last line of code sends the DatagramPacket on its way.

When the server has read all the quotes from the quote file, the while loop terminates and the run method cleans up:

```
socket.close();
```

Datagram Client

5

- Maak DatagramSocket
- Bepaal poort en adres van de server
- Maak DatagramPacket
- Verstuur vraag
- Maak leeg DatagramPacket
- Wacht op antwoord
- Sluit socket

Datagram Client

6

```
try (DatagramSocket socket = new DatagramSocket()) {
  byte[] buf = new byte[256];
  InetAddress address = InetAddress.getByName(HOST);
  DatagramPacket packet
    = new DatagramPacket(buf, buf.length, address, 4445);
  socket.send(packet); // send request

  packet = new DatagramPacket(buf, buf.length);
  socket.receive(packet); // get response
  // display response
  String received
    = new String(packet.getData(), 0, packet.getLength());
  System.out.println("Quote of the Moment: " + received);
} catch (… ex) { … }
```

Industrieel Ingenieur Informatica, UGent

The client application in this example is fairly simple. It sends a single datagram packet to the server indicating that the client would like to receive a quote of the moment. The client then waits for the server to send a datagram packet in response.

The QuoteClient class implements a client application for the QuoteServer. This application sends a request to the QuoteServer, waits for the response, and, when the response is received, displays it to the standard output. Let's look at the code in detail.

The QuoteClient class contains one method, the main method for the client application. The top of the main method declares several local variables for its use:

```
InetAddress address;
DatagramSocket socket = null;
DatagramPacket packet;
byte[] sendBuf = new byte[256];
```

Next, the main method creates a DatagramSocket:

```
DatagramSocket socket = new DatagramSocket();
```

The client uses a constructor that does not require a port number. This constructor just binds the DatagramSocket to any available local port. It doesn't matter what port the client is bound to because the DatagramPackets contain the addressing information. The server gets the port number from the DatagramPackets and send

its response to that port.

Next, the QuoteClient program sends a request to the server:

```
byte[] buf = new byte[256];
InetAddress address = InetAddress.getByName(HOST);
DatagramPacket packet = new DatagramPacket(buf, buf.length, address,
PORT);
socket.send(packet);
```

The code segment gets the Internet address for the host (presumably the name of the machine on which the server is running). This InetAddress and the port number (the port number that the server used to create its DatagramSocket) are then used to create DatagramPacket destined for that Internet address and port number. Therefore the DatagramPacket will be delivered to the quote server.
The InetAddress instance contains the address of the node (e.g. server) to send the UDP packet to. The InetAddress class represents an IP address (Internet Address). The getByName() method returns an InetAddress instance with the IP address matching the given host name.

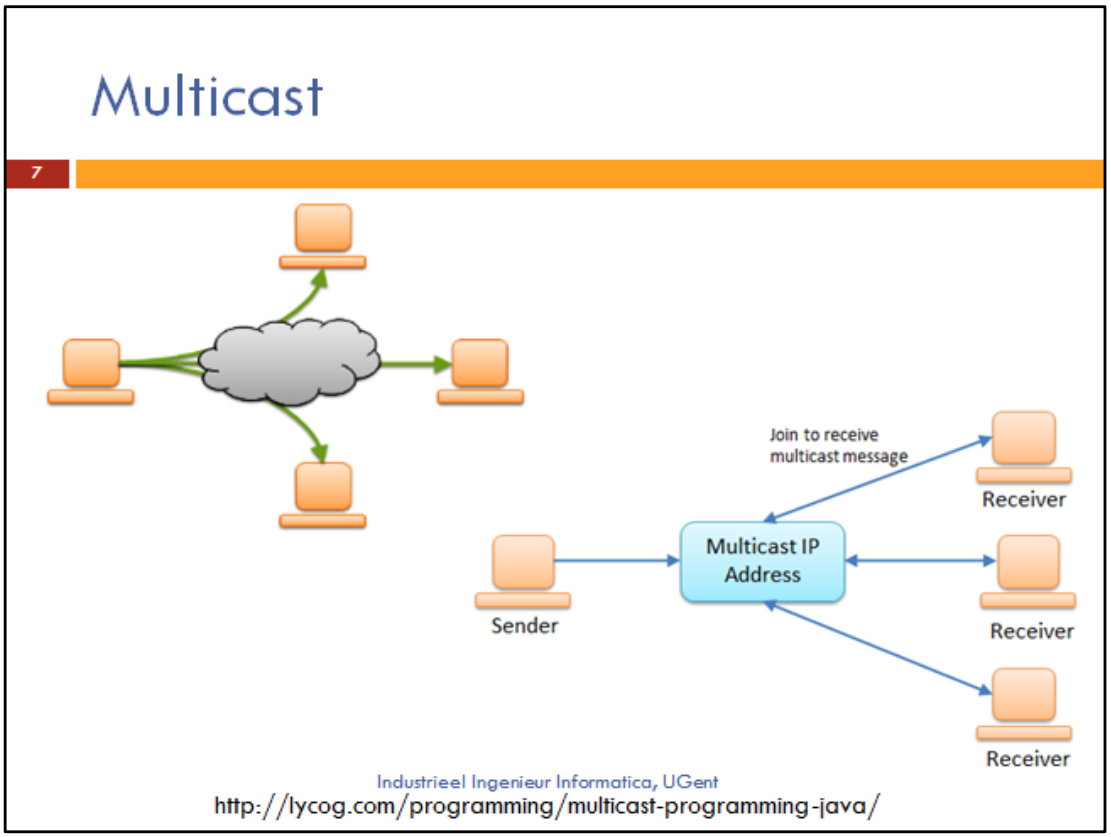
Note that the code creates a DatagramPacket with an empty byte array. The byte array is empty because this datagram packet is simply a request to the server for information. All the server needs to know to send a response--the address and port number to which reply--is automatically part of the packet.

Next, the client gets a response from the server and displays it:

```
packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
String received = new String(packet.getData(), 0, packet.getLength());
System.out.println("Quote of the Moment: " + received);
```

To get a response from the server, the client creates a "receive" packet and uses the DatagramSocket receive method to receive the reply from the server. The receive method waits until a datagram packet destined for the client comes through the socket. Note that if the server's reply is somehow lost, the client will **wait forever** because of the no-guarantee policy of the datagram model. Normally, a client sets a **timer** so that it doesn't wait forever for a reply; if no reply arrives, the timer goes off and the client retransmits.

When the client receives a reply from the server, the client uses the getData method to retrieve that data from the packet. The client then converts the data to a string and displays it.



Multicast is a special feature of UDP protocol that enable programmer to send message to a group of receivers on a specific multicast IP address and port. Multicast has advantage in this scenario.

Let us say I want to send "Hello" message to 100 computers on my network. Perhaps, my first solution is to send the "Hello" message to each of them via UDP or TCP.
What a problem is this scenario? There are 3 generic problems:

- Consume a lot of processing power on sender as it needs to send to every receiver
- Bandwidth flooding
- The arrival time is not the same for every receiver

Seeing this problem, I propose my second solution by employing Multicast. Multicast runs over UDP protocol.

Multicast Characteristics
Multicast is using UDP under the hood. So sending and receiving data are much the same as UDP
The big noticeable from UDP

- Sender should address packages to an IP number in the range between **224.0.0.1** and **239.255.255.254**. Please see the full range of Multicast IP Address
- Receivers must join multicast group to receive packet

Several multicast sockets can be bound simultaneously to the same port (Contrary to UDP and TCP)
Multicast is viable for video conference, service discovery application, etc.

Multicast Server

8

- Maak `DatagramSocket`
- Lus
  - Bepaal poort en adres van de multicast
  - Maak `DatagramPacket` met bericht
  - Verstuur pakket
  - Slaap even
- Sluit socket

Multicast Server

9

```
DatagramSocket socket = new DatagramSocket(4445);
boolean moreQuotes = true;
while (moreQuotes) {
  String dString = …; // multicast bericht
  byte[] buf = dString.getBytes();

  InetAddress group = InetAddress.getByName("230.0.0.1");
  DatagramPacket packet
    = new DatagramPacket(buf, buf.length, group, 4446);
  socket.send(packet);

  try { sleep((long) (Math.random() * FIVE_SECONDS));
   } catch (InterruptedException e) { }
}
socket.close();
```

Industrieel Ingenieur Informatica, UGent

In addition to DatagramSocket, which lets programs send packets to one another, java.net includes a class called MulticastSocket. This kind of socket is used on the client-side to listen for packets that the server broadcasts to multiple clients.

Let's rewrite the quote server so that it broadcasts DatagramPackets to multiple recipients. Instead of sending quotes to a specific client that makes a request, the new server now needs to broadcast quotes at a regular interval. The client needs to be modified so that it passively listens for quotes and does so on a MulticastSocket.

This example is comprised of three classes which are modifications of the three classes from the previous example: MulticastServer, MulticastServerThread, and MulticastClient. This discussion highlights the interesting parts of these classes.

Here is the new version of the server's main program. The differences between this code and the previous version, QuoteServer, are shown in bold:

```
import java.io.*;
public class MulticastServer {
        public static void main(String[] args) throws IOException {
                new MulticastServerThread().start();
        }
}
```

Basically, the server got a new name and creates a MulticastServerThread instead of

a QuoteServerThread. Now let's look at the MulticastServerThread which contains the heart of the server. Here's its class declaration:

```
public class MulticastServerThread extends QuoteServerThread { // ... }
```

We've made this class a subclass of QuoteServerThread so that it can use the constructor, and inherit some member variable and the getNextQuote method. Recall that QuoteServerThread creates a DatagramSocket bound to port 4445 and opens the quote file. The DatagramSocket's port number doesn't actually matter in this example because the client never send anything to the server.
The only method explicitly implemented in MulticastServerThread is its run method. The differences between this run method and the one in QuoteServerThread are shown in bold:

```
public void run() {
        while (moreQuotes) {
                try {
                        byte[] buf = new byte[256];
                        // don't wait for request...just send a quote
                        String dString = null;
                        if (in == null) dString = new Date().toString();
                        dString = getNextQuote();
                        buf = dString.getBytes();
                        InetAddress group = InetAddress.getByName("203.0.113.0");
                        DatagramPacket packet; packet = new DatagramPacket(buf,
                        buf.length, group, 4446);
                        socket.send(packet);
                        try {
                                sleep((long)Math.random() * FIVE_SECONDS);
                        } catch (InterruptedException e) { }
                } catch (IOException e) {
                                e.printStackTrace();
                                moreQuotes = false;
                }
        }
        socket.close();
}
```

The interesting change is how the DatagramPacket is constructed, in particular, the InetAddress and port used to construct the DatagramPacket. Recall that the previous example retrieved the InetAddress and port number from the packet sent to the server from the client. This was because the server needed to reply directly to the client. Now, the server needs to address multiple clients. So this time both the InetAddress and the port number are hard-coded.
The hard-coded port number is 4446 (the client must have a MulticastSocket bound to this port). The hard-coded InetAddress of the DatagramPacket is "203.0.113.0" and is a group identifier (rather than the Internet address of the machine on which a single client is running). This particular address was arbitrarily chosen from the

reserved for this purpose.
Created in this way, the DatagramPacket is destined for all clients listening to port number 4446 who are member of the "203.0.113.0" group.

Multicast Client

10

- Bepaal poort
- Maak `MulticastSocket`
- Bepaal multicastadres
- Sluit aan bij multicastgroep
- Maak lege `DatagramPacket`
- Wacht op bericht
- Verlaat multicastgroep
- Sluit socket

Multicast Client

11

```
try (MulticastSocket socket = new MulticastSocket(4446)) {
  InetAddress address = InetAddress.getByName("230.0.0.1");
  socket.joinGroup(address);

  DatagramPacket packet;
  for (int i = 0; i < 5; i++) { // get a few quotes
    byte[] buf = new byte[256];
    packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);

    String received = new String(packet.getData(), 0, packet.getLength()
    System.out.println("Quote of the Moment: " + received);
  }
  socket.leaveGroup(address);
} catch (…) { … }
```

Industriel Ingenieur Informatica, UGent

The client needs to be modified so that it passively listens for quotes and does so on a MulticastSocket.

To listen to port number 4446, the new client program just created its MulticastSocket with that port number. To become a member of the "203.0.113.0" group, the client calls the MulticastSocket's joinGroup method with the InetAddress that identifies the group. Now, the client is set up to receive DatagramPackets destined for the port and group specified. Here's the relevant code from the new client program (which was also rewritten to passively receive quotes rather than actively request them). The bold statements are the ones that interact with the MulticastSocket:

```
MulticastSocket socket = new MulticastSocket(4446);
InetAddress group = InetAddress.getByName("203.0.113.0");
socket.joinGroup(group);
DatagramPacket packet;
for (int i = 0; i < 5; i++) {
        byte[] buf = new byte[256];
        packet = new DatagramPacket(buf, buf.length);
        socket.receive(packet);
        String received = new String(packet.getData());
        System.out.println("Quote of the Moment: " + received);
}
socket.leaveGroup(group);
```

```
socket.close();
```

Notice that the server uses a DatagramSocket to broadcast packet received by the client over a MulticastSocket. Alternatively, it could have used a MulticastSocket. The socket used by the server to send the DatagramPacket is not important. What's important when broadcasting packets is the addressing information contained in the DatagramPacket, and the socket used by the client to listen for it.

MulticastSocket extends DatagramSocket.

DatagramChannel - server

12

```
// maak channel
DatagramChannel in = DatagramChannel.open();
// verbind met poort
in.socket().bind(new InetSocketAddress(PORT));
// maak buffer
ByteBuffer bufIn = ByteBuffer.allocate(48);
bufIn.clear(); // bij volgende request
// ontvang request
in.receive(bufIn);
// lees buffer
bufIn.flip();
```

Industrieel Ingenieur Informatica, UGent

A Java NIO DatagramChannel is a channel that can send and receive UDP packets. Since UDP is a connection-less network protocol, you cannot just by default read and write to a DatagramChannel like you do from other channels. Instead you send and receive packets of data.

Opening a DatagramChannel

Here is how you open a DatagramChannel:

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));
```

This example opens a DatagramChannel which can receive packets on UDP port 9999.

Receiving Data

You receive data from a DatagramChannel by calling its receive() method, like this:

```
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
channel.receive(buf);
```

The receive() method will copy the content of a received packet of data into the given Buffer. If the received packet contains more data than the Buffer can contain, the remaining data is discarded silently.

DatagramChannel - client

13

```
try (DatagramChannel channel = DatagramChannel.open()) {
  ByteBuffer buf = ByteBuffer.allocate(48);
  for (int i = 0; i < 10; i++) {
    buf.clear();
    String newData = "logbericht " + System.currentTimeMillis();
    buf.put(newData.getBytes());
    buf.flip();

    int bytesSent
      = channel.send(buf, new InetSocketAddress("localhost", 4445));
    System.out.println(bytesSent + " verzonden");
  }
}
```

Industrieel Ingenieur Informatica, UGent

A Java NIO DatagramChannel is a channel that can send and receive UDP packets. Since UDP is a connection-less network protocol, you cannot just by default read and write to a DatagramChannel like you do from other channels. Instead you send and receive packets of data.

Opening a DatagramChannel

Here is how you open a DatagramChannel:

```
DatagramChannel channel = DatagramChannel.open();
```

Sending Data

You can send data via a DatagramChannel by calling its send() method, like this:

```
String newData = "New String to write to file..." + System.currentTimeMillis();
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
int bytesSent = channel.send(buf, new InetSocketAddress("jenkov.com",
80));
```

This example sends the string to the "jenkov.com" server on UDP port 80. Nothing is listening on that port though, so nothing will happen. You will not be notified of whether the send packet was received or not, since UDP does not make any guarantees about delivery of data.

Connecting to a Specific Address

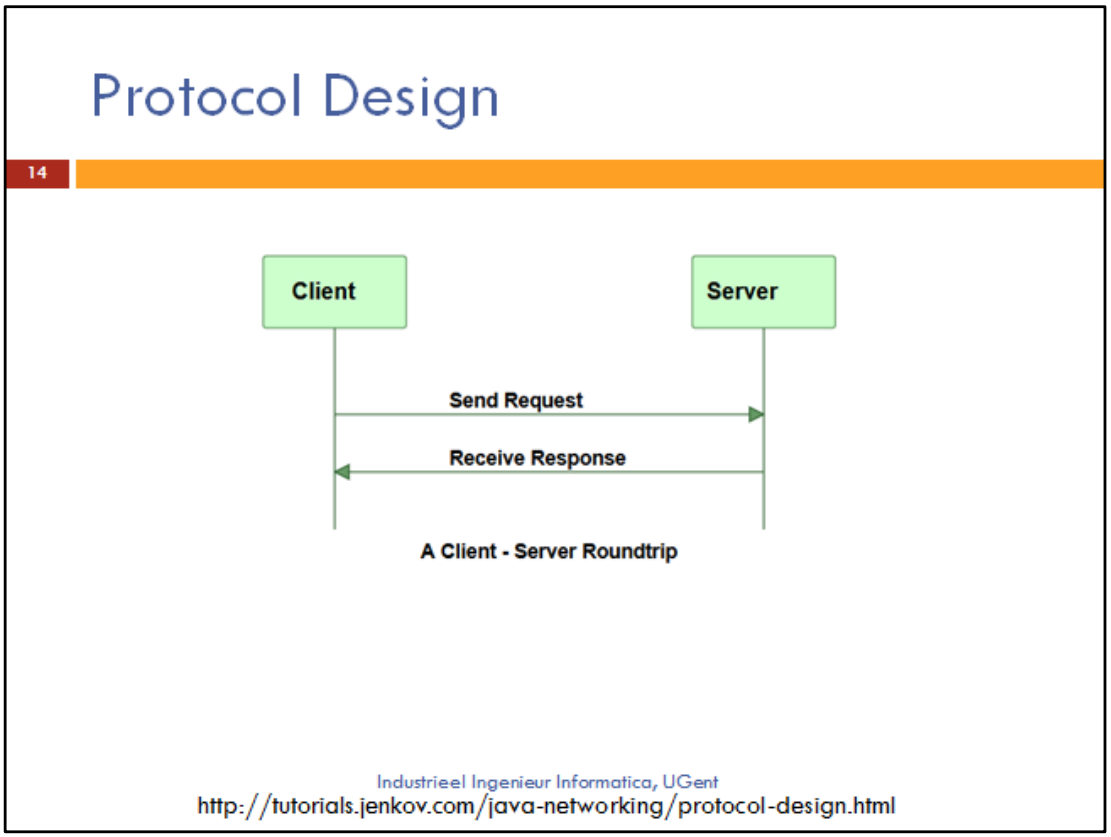
It is possible to "connect" a DatagramChannel to a specific address on the network. Since UDP is connection-less, this way of connecting to an address does not create a real connection, like with a TCP channel. Rather, it locks your DatagramChannel so you can only send and receive data packets from one specific address.

Here is an example:

```
channel.connect(new InetSocketAddress("jenkov.com", 80));
```

When connected you can also use the read() and write() method, as if you were using a traditional channel. You just don't have any guarantees about delivery of the sent data. Here are a few examples:

```
int bytesRead = channel.read(buf);
int bytesWritten = channel.write(buf);
```



If you are designing a client-server system you may also have to design a communication protocol between the client and the server. Of course, sometimes this protocol is already have been decided for you, e.g. HTTP, XML-RPC (XML over HTTP), or SOAP (also XML over HTTP). But once in a while the protocol decision is open, so let's look at a few issued you may want to think about when designing your client - server protocol:

Client - Server Roundtrips
When a client and server communicates to perform some operation they exchange information. For instance, the client will ask for a service to be performed, and the server will attempt to perform it, and send back a response telling the client of the result. Such an exchange of information between the client and server is called a **roundtrip**.

When a computer (client or server) sends data to another computer over the internet it takes some time from the time the data is sent, to the data is received at the other end. This is the time it takes the data to travel over the internet. This time is called **latency**.

The more roundtrips you have in your protocol, the slower the protocol becomes, especially if latency is high. The HTTP protocol consists of only a single request and a single response to perform its service. A single roundtrip in other words. The SMTP protocol on the other hand, consists of several roundtrips between the client and

the server before an email is sent.

The only reason to break your protocol up into multiple roundtrips is, if you have a large amount of data to send from the client to the server. You have two options in this case:

Send the header information in a separate roundtrip.
Break the message body up into smaller chunks.

Sending the header in a separate roundtrip (the first) can be smart if the server can do some initial pre-validation of e.g. header information. If that header information is invalid, sending the large body of data would have been a waste anyways.

If the network connection fails while you are transfering a large amount of data, you may have to resend all that data from scratch. By breaking the data up into smaller chunks you only have to resend the chunks from the chunk where the network connection failed and onwards. The successfully transfered chunks do not have be resent.

Demarcating the End of Requests and Responses

If your protocol allows multiple requests to be send over the same connection, you need some way for the server to know when one request ends, and a new begins. The client also needs to know when one response ends, and another begins.

You have two options for demarcating the end of a request:

Send the length in bytes of the request in the beginning of the request.
Send an end-of-request marker after the request data.

HTTP uses the first mechanism. In one of the request headers the "Content-Length" is sent. This header tells how many bytes after the headers that belongs to the request.
The advantage of this model is that you don't have the overhead of the end-of-request marker. Nor do you have to encode the body of the data to avoid the data looking like the end-of-request marker.

The disadvantage of the first method is that the sender must know how many bytes are transfered before the data is transfered. If the data is generated dynamically you will first have to buffer all the data before sending it, to count the number of bytes.

By using an end-of-request marker you don't have to know how many bytes you are sending. You just need to send an end-of-request marker at the end of the data. You do, however, have to make sure that the data sent does not contain any data that can be mistaken for the end-of-request marker. Here one way to do that:

Lets say the end-of-request marker is the byte value 255. Of course the data can contain the value 255 too. So, for each byte in the data that contains the value 255 you add an extra byte, also with the value 255. The end-of-request marker is

changed from the byte value 255 to 255 followed by the value 0. Here are the encodings summarized:

255 in data --> 255, 255
end-of-request --> 255, 0

The sequence 255, 0 can never occur in the data, since you are changing all 255's to 255,255. And, a 255,255,0 will not be mistaken for a 255,0. The first 255's will be interpreted together, and the last 0 by itself.

Penetrating Firewalls
Most firewalls block all other traffic than the HTTP protocol. Therefore it can be a good idea to layer your protocol on top of HTTP, like XML-RPC, SOAP and REST does. To layer your protocol on top of HTTP you send your data forth and back between client and server inside HTTP requests and responses. Remember, an HTTP request and response can contain more than just text or HTML. You can send binary data in there too.

The only thing that can be a little weird by layering your request on top of the HTTP protocol is that an HTTP request **must** contain a "Host" header field. If you are designing a P2P protocol on top of HTTP, your peers most likely won't be running multiple "Hosts". This required header field is in that situation an unnecessary overhead (but a small one).

WINDOWS COMMUNICATION FOUNDATION (WCF)

Veerle Ongenae

WCF - overzicht

2

- Wat is WCF
- Architectuur WCF
  - Contracten, Gedrag, Berichten, Hosting
- Ontwikkeling WCF
  - Ontwerp servicecontract
  - Implementatie contract
  - Configuratie
  - Hosting
  - Ontwikkelen client

WCF

3

- Gedistribueerde applicaties
  - Communicatie tussen applicaties
    - Op dezelfde computer
    - Over een intranet
    - Over het internet
- Service-oriented applications (~SOA)
- Webservices
  - SOAP
  - REST
- Transport over
  - Named pipes
  - TCP
  - HTTP
- Berichten
  - Binair
  - Tekst

Industrieel Ingenieur Informatica, UGent

Windows Communication Foundation (WCF) is a framework for building service-oriented applications. Using WCF, you can send data as asynchronous messages from one service endpoint to another. A service endpoint can be part of a continuously available service hosted by IIS, or it can be a service hosted in an application. An endpoint can be a client of a service that requests data from a service endpoint. The messages can be as simple as a single character or word sent as XML, or as complex as a stream of binary data. A few sample scenarios include:

A secure service to process business transactions.

A service that supplies current data to others, such as a traffic report or other monitoring service.

A chat service that allows two people to communicate or exchange data in real time.

A dashboard application that polls one or more services for data and presents it in a logical presentation.

Features WCF

Service Orientation

One consequence of using WS standards is that WCF enables you to create service oriented applications. Service-oriented architecture (SOA) is the reliance on Web services to send and receive data. The services have the general advantage of being loosely-coupled instead of hard-coded from one application to another. A loosely-coupled relationship implies that any client created on any platform can connect to any service as long as the essential contracts are met.

Interoperability

WCF implements modern industry standards for Web service interoperability.

Multiple Message Patterns

Messages are exchanged in one of several patterns. The most common pattern is the request/reply pattern, where one endpoint requests data from a second endpoint. The second endpoint replies. There are other patterns such as a one-way message in which a single endpoint sends a message without any expectation of a reply. A more complex pattern is the duplex exchange pattern where

two endpoints establish a connection and send data back and forth, similar to an instant messaging program.

Service Metadata

WCF supports publishing service metadata using formats specified in industry standards such as WSDL, XML Schema and WS-Policy. This metadata can be used to automatically generate and configure clients for accessing WCF services. Metadata can be published over HTTP and HTTPS or using the Web Service Metadata Exchange standard.

Data Contracts

Because WCF is built using the .NET Framework, it also includes code-friendly methods of supplying the contracts you want to enforce. One of the universal types of contracts is the data contract. In essence, as you code your service using Visual C# or Visual Basic, the easiest way to handle data is by creating classes that represent a data entity with properties that belong to the data entity. WCF includes a comprehensive system for working with data in this easy manner. Once you have created the classes that represent data, your service automatically generates the metadata that allows clients to comply with the data types you have designed.

Security

Messages can be encrypted to protect privacy and you can require users to authenticate themselves before being allowed to receive messages. Security can be implemented using well-known standards such as SSL or WS-SecureConversation.

Multiple Transports and Encodings

Messages can be sent on any of several built-in transport protocols and encodings. The most common protocol and encoding is to send text encoded SOAP messages using is the HyperText Transfer Protocol (HTTP) for use on the World Wide Web. Alternatively, WCF allows you to send messages over TCP, named pipes, or MSMQ. These messages can be encoded as text or using an optimized binary format. Binary data can be sent efficiently using the MTOM standard. If none of the provided transports or encodings suit your needs you can create your own custom transport or encoding.

Reliable and Queued Messages

WCF supports reliable message exchange using reliable sessions implemented over WS-Reliable Messaging and using MSMQ.

Durable Messages

A durable message is one that is never lost due to a disruption in the communication. The messages in a durable message pattern are always saved to a database. If a disruption occurs, the database allows you to resume the message exchange when the connection is restored. You can also create a durable message using the Windows Workflow Foundation (WF).

Transactions

WCF also supports transactions using one of three transaction models: WS-AtomicTtransactions, the APIs in the System.Transactions namespace, and Microsoft Distributed Transaction Coordinator.

AJAX and REST Support

REST is an example of an evolving Web 2.0 technology. WCF can be configured to process "plain" XML data that is not wrapped in a SOAP envelope. WCF can also be extended to support specific XML formats, such as ATOM (a popular RSS standard), and even non-XML formats, such as JavaScript Object Notation (JSON).

Extensibility

The WCF architecture has a number of extensibility points. If extra capability is required, there are a number of entry points that allow you to customize the behavior of a service.

WCF Concepten

4

- Client-Server
- Communicatie
  - Berichten
  - Tussen “endpoints”
- Service
  - Bestaat uit verschillende “endpoints”
- Endpoint
  - Waarnaar berichten sturen
  - Hoe berichten sturen
  - Structuur berichten

Industrieel Ingenieur Informatica, UGent

WCF is a runtime and a set of APIs for creating systems that send messages between services and clients. The same infrastructure and APIs are used to create applications that communicate with other applications on the same computer system or on a system that resides in another company and is accessed over the Internet.

Messaging and Endpoints
WCF is based on the notion of message-based communication, and anything that can be modeled as a message (for example, an HTTP request or a Message Queuing (also known as MSMQ) message) can be represented in a uniform way in the programming model. This enables a unified API across different transport mechanisms.

The model distinguishes between clients, which are applications that initiate communication, and services, which are applications that wait for clients to communicate with them and respond to that communication. A single application can act as both a client and a service.

Messages are sent between endpoints. Endpoints are places where messages are sent or received (or both), and they define all the information required for the message exchange. A service exposes one or more application endpoints (as well as zero or more infrastructure endpoints), and the client generates an endpoint that is compatible with one of the service's endpoints.

An endpoint describes in a standard-based way where messages should be sent, how they should be sent, and what the messages should look like. A service can expose this information as metadata that clients can process to generate appropriate WCF clients and communication stacks.

Communication Protocols
One required element of the communication stack is the transport protocol. Messages can be sent over intranets and the Internet using common transports, such as HTTP and TCP. Other transports are included that support communication with Message Queuing applications and nodes on a Peer

Networking mesh.

Another required element in the communication stack is the encoding that specifies how any given message is formatted. WCF provides the following encodings:

Text encoding, an interoperable encoding.
Message Transmission Optimization Mechanism (MTOM) encoding, which is an interoperable way for efficiently sending unstructured binary data to and from a service.
Binary encoding for efficient transfer.

Message Patterns
WCF supports several messaging patterns, including request-reply, one-way, and duplex communication. Different transports support different messaging patterns, and thus affect the types of interactions that they support. The WCF APIs and runtime also help you to send messages securely and reliably.

message
A self-contained unit of data that can consist of several parts, including a body and headers.

service
A construct that exposes one or more endpoints, with each endpoint exposing one or more service operations.

endpoint
A construct at which messages are sent or received (or both). It comprises a location (an address) that defines where messages can be sent, a specification of the communication mechanism (a binding) that described how messages should be sent, and a definition for a set of messages that can be sent or received (or both) at that location (a service contract) that describes what message can be sent.

A WCF service is exposed to the world as a collection of endpoints.
application endpoint

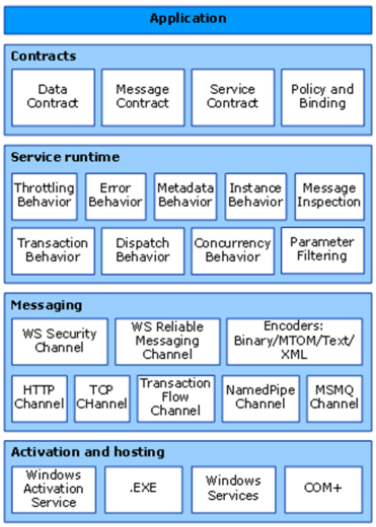
An endpoint exposed by the application and that corresponds to a service contract implemented by the application.

WCF - overzicht

5

- Wat is WCF
- Architectuur WCF
  - Contracten, Gedrag, Berichten, Hosting
- Ontwikkeling WCF
  - Ontwerp servicecontract
  - Implementatie contract
  - Configuratie
  - Hosting
  - Ontwikkelen client

WCF-architectuur



https://msdn.microsoft.com/en-us/library/ms733128%28v=vs.110%29.aspx

WCF Architectuur - Contracten

7

- Data Contract
  - Parameters in bericht
  - XSD
- Message Contract
  - Structuur bericht
- Service Contract
  - Beschikbare interface
  - Signatuur methodes
- Bindings, Policies
  - Hoe communiceren met service?
  - Beveiliging?

Industrieel Ingenieur Informatica, UGent

Contracts and Descriptions
Contracts define various aspects of the message system. The data contract describes every parameter that makes up every message that a service can create or consume. The message parameters are defined by XML Schema definition language (XSD) documents, enabling any system that understands XML to process the documents. The message contract defines specific message parts using SOAP protocols, and allows finer-grained control over parts of the message, when interoperability demands such precision. The service contract specifies the actual method signatures of the service, and is distributed as an interface in one of the supported programming languages, such as Visual Basic or Visual C#.
Policies and bindings stipulate the conditions required to communicate with a service. For example, the binding must (at a minimum) specify the transport used (for example, HTTP or TCP), and an encoding. Policies include security requirements and other conditions that must be met to communicate with a service.

service operation
A procedure defined in a service's code that implements the functionality for an operation. This operation is exposed to clients as methods on a WCF client. The method can return a value, and can take an optional number of arguments, or take no arguments, and return no response. For example, an operation that functions as a simple "Hello" can be used as a notification of a client's presence and to begin a series of operations.

service contract
Ties together multiple related operations into a single functional unit. The contract can define service-level settings, such as the namespace of the service, a corresponding callback contract, and other such settings. In most cases, the contract is defined by creating an interface in the programming language of your choice and applying the ServiceContractAttribute attribute to the interface. The actual service code results by implementing the interface.

operation contract

An operation contract defines the parameters and return type of an operation. When creating an interface that defines the service contract, you signify an operation contract by applying the OperationContractAttribute attribute to each method definition that is part of the contract. The operations can be modeled as taking a single message and returning a single message, or as taking a set of types and returning a type. In the latter case, the system will determine the format for the messages that need to be exchanged for that operation.

message contract

Describes the format of a message. For example, it declares whether message elements should go in headers versus the body, what level of security should be applied to what elements of the message, and so on.

fault contract

Can be associated with a service operation to denote errors that can be returned to the caller. An operation can have zero or more faults associated with it. These errors are SOAP faults that are modeled as exceptions in the programming model.

data contract

The descriptions in metadata of the data types that a service uses. This enables others to interoperate with the service. The data types can be used in any part of a message, for example, as parameters or return types. If the service is using only simple types, there is no need to explicitly use data contracts.

WCF Architectuur – gedrag

8

- Service runtime
- Eigenlijke gedrag applicatie
  - Aantal berichten dat verwerkt kan worden
  - Wat bij fouten?
  - Welke metadata beschikbaar voor de buitenwereld?
  - Hoeveel instanties tegelijk van service?
  - Transacties
  - …

Industrieel Ingenieur Informatica, UGent

Service Runtime
The service runtime layer contains the behaviors that occur only during the actual operation of the service, that is, the runtime behaviors of the service. Throttling controls how many messages are processed, which can be varied if the demand for the service grows to a preset limit. An error behavior specifies what occurs when an internal error occurs on the service, for example, by controlling what information is communicated to the client. (Too much information can give a malicious user an advantage in mounting an attack.) Metadata behavior governs how and whether metadata is made available to the outside world. Instance behavior specifies how many instances of the service can be run (for example, a singleton specifies only one instance to process all messages). Transaction behavior enables the rollback of transacted operations if a failure occurs. Dispatch behavior is the control of how a message is processed by the WCF infrastructure.

Extensibility enables customization of runtime processes. For example, message inspection is the facility to inspect parts of a message, and parameter filtering enables preset actions to occur based on filters acting on message headers.

WCF Architectuur - Berichten

9

- Verschillende kanalen verwerken berichten
  - Zowel hoofdingen als berichten
    - Bv. Kanaal voor authenticatie
- Twee type kanalen
  - Transport
    - Lezen en schrijven van berichten van en naar het netwerk
  - Protocol
    - Lezen en schrijven van hoofdingen
    - …

Industrieel Ingenieur Informatica, UGent

Messaging
The messaging layer is composed of channels. A channel is a component that processes a message in some way, for example, by authenticating a message. A set of channels is also known as a channel stack. Channels operate on messages and message headers. This is different from the service runtime layer, which is primarily concerned about processing the contents of message bodies.

There are two types of channels: transport channels and protocol channels.

Transport channels read and write messages from the network (or some other communication point with the outside world). Some transports use an encoder to convert messages (which are represented as XML Infosets) to and from the byte stream representation used by the network. Examples of transports are HTTP, named pipes, TCP, and MSMQ. Examples of encodings are XML and optimized binary.

Protocol channels implement message processing protocols, often by reading or writing additional headers to the message. Examples of such protocols include WS-Security and WS-Reliability.

The messaging layer illustrates the possible formats and exchange patterns of the data. WS-Security is an implementation of the WS-Security specification enabling security at the message layer. The WS-Reliable Messaging channel enables the guarantee of message delivery. The encoders present a variety of encodings that can be used to suit the needs of the message. The HTTP channel specifies that the HyperText Transport Protocol is used for message delivery. The TCP channel similarly specifies the TCP protocol. The Transaction Flow channel governs transacted message patterns. The Named Pipe channel enables interprocess communication. The MSMQ channel enables interoperation with MSMQ applications.

WCF Architectuur - Hosting

10

- Een service is een programma
- Moet uitgevoerd worden
  - Als executable
    - Self-hosted service
  - Gehost in een externe omgeving
    - Webserver
    - Als Windows service

Industrieel Ingenieur Informatica, UGent

Hosting and Activation

In its final form, a service is a program. Like other programs, a service must be run in an executable. This is known as a *self-hosted* service.

Services can also be *hosted*, or run in an executable managed by an external agent, such as IIS or Windows Activation Service (WAS). WAS enables WCF applications to be activated automatically when deployed on a computer running WAS. Services can also be manually run as executables (.exe files). A service can also be run automatically as a Windows service. COM+ components can also be hosted as WCF services.

Voorbeeld

11

- Service op IIS
  - tijdService
  - tijdClient
  - tijdWebserviceClient
- Service als console-applicatie
  - tijdServer
  - toonTijdVanConsole (Client)

Overzicht

12

- Ontwerp servicecontract
- Implementatie contract
- Configuratie
- Hosting
- Ontwikkelen client

Industrieel Ingenieur Informatica, UGent

Windows Communication Foundation (WCF) enables applications to communicate whether they are on the same computer, across the Internet, or on different application platforms.

The Basic Tasks
The basic tasks to perform are, in order:

Define the service contract. A service contract specifies the signature of a service, the data it exchanges, and other contractually required data.

Implement the contract. To implement a service contract, create a class that implements the contract and specify custom behaviors that the runtime should have.

Configure the service by specifying endpoints and other behavior information.

Host the service.

Build a client application.

Ontwerp servicecontract

13

- Service – dienst
  - Bestaat uit operations – opdrachten
    - Methode
      - Attribuut OperationContract
      - Parameters en teruggeefwaarde
        - Serialiseerbaar
        - Geen referenties, maar kopies
    - Eigen datatypes
      - Contract
        - Attribuut DataContract
        - Attribuut DataMember
  - Interface
    - Attribuut ServiceContract
- Namespace: System.ServiceModel;

Industrieel Ingenieur Informatica, UGent

A service contract specifies what an endpoint communicates to the outside world. At a more concrete level, it is a statement about a set of specific messages organized into basic message exchange patterns (MEPs), such as request/reply, one-way, and duplex. If a service contract is a logically related set of message exchanges, a service operation is a single message exchange. For example, a Hello operation must obviously accept one message (so the caller can announce the greeting) and may or may not return a message (depending upon the courtesy of the operation).

Service Contracts

A service contract specifies the following:
The operations a contract exposes.
The signature of the operations in terms of messages exchanged.
The data types of these messages.
The location of the operations.
The specific protocols and serialization formats that are used to support successful communication with the service.

For example, a purchase order contract might have a CreateOrder operation that accepts an input of order information types and returns success or failure information, including an order identifier. It might also have a GetOrderStatus operation that accepts an order identifier and returns order status information. A service contract of this sort would specify:
That the purchase order contract consisted of CreateOrder and GetOrderStatus operations.
That the operations have specified input messages and output messages.
The data that these messages can carry.
Categorical statements about the communication infrastructure necessary to successfully process the messages. For example, these details include whether and what forms of security are required to establish successful communication.

To convey this kind of information to other applications on many platforms (including non-Microsoft platforms), XML service contracts are publicly expressed in standard XML formats, such as Web Services Description Language (WSDL) and XML Schema (XSD), among others. Developers for many platforms can use this public contract information to create applications that can communicate with the service, both because they understand the language of the specification and because those languages are designed to enable interoperation by describing the public forms, formats, and protocols that the service supports. For more information about how WCF handles this kind of information, see Metadata.

Contracts can be expressed many ways, and while WSDL and XSD are excellent languages to describe services in an accessible way, they are difficult languages to use directly and are merely descriptions of a service, not service contract implementations. Therefore, WCF applications use managed attributes, interfaces, and classes both to define the structure of a service and to implement it.

The resulting contract defined in managed types can be exported as metadata—WSDL and XSD—when needed by clients or other service implementers. The result is a straightforward programming model that can be described (using public metadata) to any client application. The details of the underlying SOAP messages, the transportation and security-related information, and so on, can be left to WCF, which performs the necessary conversions to and from the service contract type system to the XML type system automatically.

A service contract groups the operations; specifies the message exchange pattern, message types, and data types those messages carry; and indicates categories of run-time behavior an implementation must have to support the contract (for example, it may require that messages be encrypted and signed). The service contract itself does not specify precisely how these requirements are met, only that they must be. The type of encryption or the manner in which a message is signed is up to the implementation and configuration of a compliant service.

Notice the way that the contract requires certain things of the service contract implementation and the run-time configuration to add behavior. The set of requirements that must be met to expose a service for use builds on the preceding set of requirements. If a contract makes requirements of the implementation, an implementation can require yet more of the configuration and bindings that enable the service to run. Finally, the host application must also support any requirements that the service configuration and bindings add.

Creating a Service Contract

Services expose a number of operations. In Windows Communication Foundation (WCF) applications, define the operations by creating a method and marking it with the OperationContractAttribute attribute. Then, to create a service contract, group together your operations, either by declaring them within an interface marked with the ServiceContractAttribute attribute, or by defining them in a class marked with the same attribute.

Any methods that do not have a OperationContractAttribute attribute are not service operations and are not exposed by WCF services.

Classes or Interfaces

Both classes and interfaces represent a grouping of functionality and, therefore, both can be used to define a WCF service contract. However, it is recommended that you use interfaces because they directly model service contracts. Without an implementation, interfaces do no more than define a grouping of methods with certain signatures. Implement a service contract interface and you have implemented a WCF service.

All the benefits of managed interfaces apply to service contract interfaces:

Service contract interfaces can extend any number of other service contract interfaces.

A single class can implement any number of service contracts by implementing those service contract interfaces.

You can modify the implementation of a service contract by changing the interface implementation, while the service contract remains the same.

You can version your service by implementing the old interface and the new one. Old clients connect to the original version, while newer clients can connect to the newer version.

You can, however, use a class to define a service contract and implement that contract at the same time. The advantage of creating your services by applying ServiceContractAttribute and OperationContractAttribute directly to the class and the methods on the class, respectively, is speed and simplicity. The disadvantages are that managed classes do not support multiple inheritance, and as a result they can only implement one service contract at a time. In addition, any modification to the class or method signatures modifies the public contract for that service, which can prevent unmodified clients from using your service.

Parameters and Return Values

Each operation has a return value and a parameter, even if these are void. However, unlike a local method, in which you can pass references to objects from one object to another, service operations do not pass references to objects. Instead, they pass copies of the objects.

This is significant because each type used in a parameter or return value must be serializable; that is, it must be possible to convert an object of that type into a stream of bytes and from a stream of bytes into an object.

Primitive types are serializable by default, as are many types in the .NET Framework.

The value of the parameter names in the operation signature are part of the contract and are case sensitive. If you want to use the same parameter name locally but modify the name in the published metadata, see the System.ServiceModel.MessageParameterAttribute.

Data Contracts

Service-oriented applications like Windows Communication Foundation (WCF) applications are designed to interoperate with the widest possible number of client applications on both Microsoft and non-Microsoft platforms. For the widest possible interoperability, it is recommended that you mark your types with the DataContractAttribute and DataMemberAttribute attributes to create a data contract, which is the portion of the service contract that describes the data that your service operations exchange.

Data contracts are opt-in style contracts: No type or data member is serialized unless you explicitly apply the data contract attribute. Data contracts are unrelated to the access scope of the managed code: Private data members can be serialized and sent elsewhere to be accessed publicly. WCF handles the definition of the underlying SOAP messages that enable the operation's functionality as well as the serialization of your data types into and out of the body of the messages. As long as your data types are serializable, you do not need to think about the underlying message exchange infrastructure when designing your operations.

Although the typical WCF application uses the DataContractAttribute and DataMemberAttribute attributes to create data contracts for operations, you can use other serialization mechanisms. The standard ISerializable, SerializableAttribute and IXmlSerializable mechanisms all work to handle the serialization of your data types into the underlying SOAP messages that carry them from one application to another. You can employ more serialization strategies if your data types require special support.

Servicecontract

- Interface met attributen

```
[ServiceContract(Namespace = "http://Microsoft.ServiceModel.Samples")]
public interface ICalculator {
    [OperationContract]
    double Add(double n1, double n2);
    [OperationContract]
    double Subtract(double n1, double n2);
    [OperationContract]
    double Multiply(double n1, double n2);
    [OperationContract]
    double Divide(double n1, double n2);
}
```

Type communicatie

15

- Vraag/antwoord
  - Standaard
- Eénrichtingscommunicatie (one-way contract)
  - Client → server
    - Client verwacht geen antwoord
  - Methode zonder returnwaarde
  - Geen uitvoerparameters
- Tweerichtingscommunicatie (duplex contract)
  - Client → server
    - Client verwacht geen antwoord
  - Server → client
    - Events op client
    - Informatie opvragen van client

Industrieel Ingenieur Informatica, UGent

Mapping Parameters and Return Values to Message Exchanges
Service operations are supported by an underlying exchange of SOAP messages that transfer application data back and forth, in addition to the data required by the application to support certain standard security, transaction, and session-related features. Because this is the case, the signature of a service operation dictates a certain underlying message exchange pattern (MEP) that can support the data transfer and the features an operation requires. You can specify three patterns in the WCF programming model: request/reply, one-way, and duplex message patterns.

Request/Reply
A request/reply pattern is one in which a request sender (a client application) receives a reply with which the request is correlated. This is the default MEP because it supports an operation in which one or more parameters are passed to the operation and a return value is passed back to the caller.

Note that unless you specify a different underlying message pattern, even service operations that return void are request/reply message exchanges. The result for your operation is that unless a client invokes the operation asynchronously, the client stops processing until the return message is received, even though that message is empty in the normal case.

This can slow client performance and responsiveness if the operation takes a long time to perform, but there are advantages to request/reply operations even when they return void. The most obvious one is that SOAP faults can be returned in the response message, which indicates that some service-related error condition has occurred, whether in communication or processing. SOAP faults that are specified in a service contract are passed to the client application as a FaultException<TDetail> object, where the type parameter is the type specified in the service contract. This makes notifying clients about error conditions in WCF services easy.

Eénrichtingscommunicatie

16

- Methode
  - Geen returnwaarde
  - Geen uitvoerparameters
  - Attribuut OperationContract
    - Eigenschap IsOneWay op true

```
[OperationContract (IsOneWay=true)]
void Hello(string greeting);
```

Industrieel Ingenieur Informatica, UGent

One-way
If the client of a WCF service application should not wait for the operation to complete and does not process SOAP faults, the operation can specify a one-way message pattern. A one-way operation is one in which a client invokes an operation and continues processing after WCF writes the message to the network. Typically this means that unless the data being sent in the outbound message is extremely large the client continues running almost immediately (unless there is an error sending the data). This type of message exchange pattern supports event-like behavior from a client to a service application.

A message exchange in which one message is sent and none are received cannot support a service operation that specifies a return value other than void; in this case an InvalidOperationException exception is thrown.

No return message also means that there can be no SOAP fault returned to indicate any errors in processing or communication. (Communicating error information when operations are one-way operations requires a duplex message exchange pattern.)

To specify a one-way message exchange for an operation that returns void, set the IsOneWay property to true.

Duplex communicatie

17

- Tweerichtingsverkeer
  - Geen vraag/antwoord
  - Communicatie kan in beide richtingen
    - Client start communicatie op (sessie)
      - Houdt kanaal open
    - Server kan later antwoorden
- Twee interfaces
  - Client → Server
  - Server → Client
    - Call back contract

Industrieel Ingenieur Informatica, UGent

Duplex
A duplex pattern is characterized by the ability of both the service and the client to send messages to each other independently whether using one-way or request/reply messaging. This form of two-way communication is useful for services that must communicate directly to the client or for providing an asynchronous experience to either side of a message exchange, including event-like behavior.

The duplex pattern is slightly more complex than the request/reply or one-way patterns because of the additional mechanism for communicating with the client.

To design a duplex contract, you must also design a callback contract and assign the type of that callback contract to the CallbackContract property of the ServiceContractAttribute attribute that marks your service contract.

To implement a duplex pattern, you must create a second interface that contains the method declarations that are called on the client.

When a service receives a duplex message, it looks at the ReplyTo element in that incoming message to determine where to send the reply. If the channel that is used to receive the message is not secured, then an untrusted client could send a malicious message with a target machine's ReplyTo, leading to a denial of service (DOS) of that target machine.

Duplex communicatie - syntax


```
[ServiceContract(Namespace =
    "http://Microsoft.ServiceModel.Samples",
    SessionMode=SessionMode.Required,
    CallbackContract=typeof(ICalculatorDuplex
    Callback))]
public interface ICalculatorDuplex {
    [OperationContract(IsOneWay = true)]
    void Clear();
    [OperationContract(IsOneWay = true)]
    void AddTo(double n);
    [OperationContract(IsOneWay = true)]
    void SubtractFrom(double n);
    [OperationContract(IsOneWay = true)]
    void MultiplyBy(double n);
    [OperationContract(IsOneWay = true)]
    void DivideBy(double n);
}
```

Implementatie op server

```
public interface ICalculatorDuplexCallback
    {
    [OperationContract(IsOneWay = true)]
    void Result(double result);
    [OperationContract(IsOneWay = true)]
    void Equation(string eqn);
}
```

Implementatie op client

Overzicht

- Ontwerp servicecontract
- Implementatie contract
- Configuratie
- Hosting
- Ontwikkelen client

Implementatie servicecontract

20

- Klasse die interface implementeert

```
[ServiceContract]
public interface IMath {
    [OperationContract]
    double Add(double A, double B);
    [OperationContract]
    double Multiply (double A, double B);
}
public class MathService : IMath {
    public double Add (double A, double B) { return A + B; }
    public double Multiply (double A, double B) { return A * B; }
}
```

Industrieel Ingenieur Informatica, UGent

A service is a class that exposes functionality available to clients at one or more endpoints. To create a service, write a class that implements a Windows Communication Foundation (WCF) contract. You can do this in one of two ways. You can define the contract separately as an interface and then create a class that implements that interface. Alternatively, you can create the class and contract directly by placing the ServiceContractAttribute attribute on the class itself and the OperationContractAttribute attribute on the methods available to the clients of the service.

Overzicht

- Ontwerp servicecontract
- Implementatie contract
- Configuratie
- Hosting
- Ontwikkelen client

Configuratie

22

- Specifiëren endpoint
  - Welke service?
    - Interface
    - Implementatie
  - Waar?
    - URL
  - Hoe communiceren? (binding)
    - TCP? HTTP?
    - Tekst? Binair?
    - Beveiligd?
- Configuratie = belangrijk deel WCF-programmatie

Industrieel Ingenieur Informatica, UGent

Once you have designed and implemented your service contract, you are ready to configure your service. This is where you define and customize how your service is exposed to clients, including specifying the address where it can be found, the transport and message encoding it uses to send and receive messages, and the type of security it requires.

Specifiëren endpoint

23

- Meestal in configuratie
- Kan ook in code

Industrieel Ingenieur Informatica, UGent

configuration versus coding

Control of an application can be done either through coding, through configuration, or through a combination of both. Configuration has the advantage of allowing someone other than the developer (for example, a network administrator) to set client and service parameters after the code is written and without having to recompile. Configuration not only enables you to set values like endpoint addresses, but also allows further control by enabling you to add endpoints, bindings, and behaviors. Coding allows the developer to retain strict control over all components of the service or client, and any settings done through the configuration can be inspected and if needed overridden by the code.

Specifiëren enpoint in code

Adres

```
Uri baseAddress = new Uri("http://localhost:8000/ServiceModelSamples/Service");
```

Implementatie service

```
ServiceHost ServiceHost selfHost = new ServiceHost(typeof(CalculatorService),
    baseAddress);
try {
    selfHost.AddServiceEndpoint( typeof(ICalculator), new WSHttpBinding(),
    "CalculatorService");
    … // gedrag instellen, service runnen
} catch (CommunicationException ce) { Console.WriteLine("An exception occurred:
    {0}", ce.Message); selfHost.Abort();
}
```

Welke service? Binding

Specifiëren endpoint in configuratie

25

```
<system.ServiceModel>
    <services>
        <service name="Microsoft.ServiceModel.Samples.CalculatorService">
            <endpoint address="http://localhost:8000/ServiceModelSamples/Service "
    binding="basicHttpBinding" bindingConfiguration="myBindingConfiguration1"
    contract="Microsoft.ServiceModel.Samples.ICalculator" />
        </service>
    </services>
    <bindings>
        <basicHttpBinding> <binding name="myBindingConfiguration1" closeTimeout="00:01:00" />
        </basicHttpBinding>  </bindings>
    <behaviors>
        <serviceBehaviors> <behavior> <serviceMetadata httpGetEnabled="True"/>
        <serviceDebug includeExceptionDetailInFaults="False" /> </behavior> </serviceBehaviors>
    </behaviors>
</system.ServiceModel>
```

Klassenaam implementatie

URL

Binding

Interface

Opvragen metadata service

Gedrag service

Industrieel Ingenieur Informatica, UGent

Configuring a Windows Communication Foundation (WCF) service with a configuration file gives you the flexibility of providing endpoint and service behavior data at the point of deployment instead of at design time.

A WCF service is configurable using the .NET Framework configuration technology. Most commonly, XML elements are added to the Web.config file for an Internet Information Services (IIS) site that hosts a WCF service. The elements allow you to change details such as the endpoint addresses (the actual addresses used to communicate with the service) on a machine-by-machine basis. In addition, WCF includes several system-provided elements that allow you to quickly select the most basic features for a service. Starting with .NET Framework version 4, WCF comes with a new default configuration model that simplifies WCF configuration requirements. If you do not provide any WCF configuration for a particular service, the runtime automatically configures your service with some standard endpoints and default binding/behavior. In practice, writing configuration is a major part of programming WCF applications.

When configuring a service in Visual Studio, use either a Web.config file or an App.config file to specify the settings. The choice of the configuration file name is determined by the hosting environment you choose for the service. If you are using IIS to host your service, use a Web.config file. If you are using any other hosting environment, use an App.config file.

In WCF service configuration files, the <**system.serviceModel**> section contains a <**service**> element for each service hosted. The <**service**> element contains a collection of <**endpoint**> elements that specify the endpoints exposed for each service and optionally a set of service behaviors.

Each service has these attributes:
name. Specifies the type that provides an implementation of a service contract. This is a fully qualified name which consists of the namespace, a period, and then the type name.
behaviorConfiguration. Specifies the name of one of the **behavior** elements found in the **behaviors**

element. The specified behavior governs actions such as whether the service allows impersonation. If its value is the empty name or no **behaviorConfiguration** is provided then the default set of service behaviors is added to the service.

The <**endpoint**> elements specify the address, binding, and contract exposed by the endpoint, and optionally binding configuration and endpoint behaviors. They are represented by the following attributes:

address. Specifies the service's Uniform Resource Identifier (URI), which can be an absolute address or one that is given relative to the base address of the service.
binding. Typically specifies a system-provided binding, but can also specify a user-defined binding. The binding specified determines the type of transport, security and encoding used, and whether reliable sessions, transactions, or streaming is supported or enabled.
bindingConfiguration. If the default values of a binding must be modified, this can be done by configuring the appropriate **binding** element in the **bindings** element. This attribute should be given the same value as the **name** attribute of the **binding** element that is used to change the defaults. If no name is given, or no **bindingConfiguration** is specified in the binding, then the default binding of the binding type is used in the endpoint.
contract. Specifies the interface that defines the contract. This is the interface implemented in the common language runtime (CLR) type specified by the **name** attribute of the **service** element.

The <**system.serviceModel**> section also contains a <**behaviors**> element that allows you to specify service or endpoint behaviors.

infrastructure endpoint
An endpoint that is exposed by the infrastructure to facilitate functionality that is needed or provided by the service that does not relate to a service contract. For example, a service might have an infrastructure endpoint that provides metadata information.

address
Specifies the location where messages are received. It is specified as a Uniform Resource Identifier (URI). The URI schema part names the transport mechanism to use to reach the address, such as HTTP and TCP. The hierarchical part of the URI contains a unique location whose format is dependent on the transport mechanism.

The endpoint address enables you to create unique endpoint addresses for each endpoint in a service or, under certain conditions, to share an address across endpoints. The following example shows an address using the HTTPS protocol with a non-default port:
HTTPS://cohowinery:8005/ServiceModelSamples/CalculatorService

binding
Defines how an endpoint communicates to the world. It is constructed of a set of components called binding elements that "stack" one on top of the other to create the communication infrastructure. At the very least, a binding defines the transport (such as HTTP or TCP) and the encoding being used (such as text or binary). A binding can contain binding elements that specify details like the security mechanisms used to secure messages, or the message pattern used by an endpoint.

binding element
Represents a particular piece of the binding, such as a transport, an encoding, an implementation of an infrastructure-level protocol (such as WS-ReliableMessaging), or any other component of the communication stack.

behaviors
A component that controls various run-time aspects of a service, an endpoint, a particular

operation, or a client. Behaviors are grouped according to scope: common behaviors affect all endpoints globally, service behaviors affect only service-related aspects, endpoint behaviors affect only endpoint-related properties, and operation-level behaviors affect particular operations. For example, one service behavior is throttling, which specifies how a service reacts when an excess of messages threaten to overwhelm its handling capabilities. An endpoint behavior, on the other hand, controls only aspects that are relevant to endpoints, such as how and where to find a security credential.

system-provided bindings

WCF includes a number of system-provided bindings. These are collections of binding elements that are optimized for specific scenarios. For example, the WSHttpBinding is designed for interoperability with services that implement various WS-* specifications. These predefined bindings save time by presenting only those options that can be correctly applied to the specific scenario. If a predefined binding does not meet your requirements, you can create your own custom binding.

channel

A concrete implementation of a binding element. The binding represents the configuration, and the channel is the implementation associated with that configuration. Therefore, there is a channel associated with each binding element. Channels stack on top of each other to create the concrete implementation of the binding: the channel stack.

security

In WCF, includes confidentiality (encryption of messages to prevent eavesdropping), integrity (the means for detection of tampering with the message), authentication (the means for validation of servers and clients), and authorization (the control of access to resources). These functions are provided by either leveraging existing security mechanisms, such as TLS over HTTP (also known as HTTPS), or by implementing one or more of the various WS-* security specifications.

transport security mode

Specifies that confidentiality, integrity, and authentication are provided by the transport layer mechanisms (such as HTTPS). When using a transport like HTTPS, this mode has the advantage of being efficient in its performance, and well understood because of its prevalence on the Internet. The disadvantage is that this kind of security is applied separately on each hop in the communication path, making the communication susceptible to a "man in the middle" attack.

message security mode

Specifies that security is provided by implementing one or more of the security specifications, such as the specification named Web Services Security: SOAP Message Security. Each message contains the necessary mechanisms to provide security during its transit, and to enable the receivers to detect tampering and to decrypt the messages. In this sense, the security is encapsulated within every message, providing end-to-end security across multiple hops. Because security information becomes part of the message, it is also possible to include multiple kinds of credentials with the message (these are referred to as claims). This approach also has the advantage of enabling the message to travel securely over any transport, including multiple transports between its origin and destination. The disadvantage of this approach is the complexity of the cryptographic mechanisms employed, resulting in performance implications.

transport with message credential security mode

Specifies the use of the transport layer to provide confidentiality, authentication, and integrity of the messages, while each of the messages can contain multiple credentials (claims) required by the receivers of the message.

WS-*

Shorthand for the growing set of Web Service (WS) specifications, such as WS-Security, WS-ReliableMessaging, and so on, that are implemented in WCF.

Bestaande bindingen

26

- Voorzien in WCF
- Een aantal voorbeelden
  - BasicHttpBinding
    - HTTP
    - Tekst/XML
    - Bv. Asmx
  - WSHttpBinding
    - Beveiligd, niet-duplex
  - NetTcpBinding
    - Beveiligd, communicatie tussen WCF-applicaties op verschillende machines
  - NetNamedPipeBinding
    - Beveiligd, geoptimaliseerd tussen WCF-applicaties op één machine

Industrieel Ingenieur Informatica, UGent

Bindings are objects that are used to specify the communication details that are required to connect to the endpoint of a Windows Communication Foundation (WCF) service. Each endpoint in a WCF service requires a binding to be well-specified. This topic outlines the types of communication details that the bindings define, the elements of a binding, what bindings are included in WCF, and how a binding can be specified for an endpoint.

Bindings are objects that are used to specify the communication details that are required to connect to the endpoint of a Windows Communication Foundation (WCF) service. Each endpoint in a WCF service requires a binding to be well-specified. This topic outlines the types of communication details that the bindings define, the elements of a binding, what bindings are included in WCF, and how a binding can be specified for an endpoint.

What a Binding Defines
The information in a binding can be very basic, or very complex. The most basic binding specifies only the transport protocol (such as HTTP) that must be used to connect to the endpoint. More generally, the information a binding contains about how to connect to an endpoint falls into one of the following categories.

Protocols Determines the security mechanism being used: either reliable messaging capability or transaction context flow settings.
Encoding Determines the message encoding (for example, text or binary).
Transport Determines the underlying transport protocol to use (for example, TCP or HTTP).

System-Provided Bindings
The information in a binding can be complex, and some settings may not be compatible with others. For this reason, WCF includes a set of system-provided bindings. These bindings are designed to cover most application requirements. The following classes represent some examples of system-provided bindings:

BasicHttpBinding: An HTTP protocol binding suitable for connecting to Web services that conforms to the WS-I Basic Profile specification (for example, ASP.NET Web services-based services).
WSHttpBinding: An interoperable binding suitable for connecting to endpoints that conform to the WS-* protocols.
NetNamedPipeBinding: Uses the .NET Framework to connect to other WCF endpoints on the same machine.
NetMsmqBinding: Uses the .NET Framework to create queued message connections with other WCF endpoints.

basicHttpBinding → adres start met “http”
netTcpBinding → adres start met “net.tcp”
netNamedPipeBinding → adres start met “net.pipe”
netMSMQBinding → adres start met “net.msmq”

Overzicht

- Ontwerp servicecontract
- Implementatie contract
- Configuratie
- Hosting
- Ontwikkelen client

Hosting

28

- Publiceren
  - Runtime omgeving
    - Beheren levensloop en context
- Windows proces
  - Ondersteunen managed code
  - Voorbeeld
    - IIS
    - Console-applicatie (zelf "hosten")
    - Windows service
    - …
- Code service onafhankelijk van host

Industrieel Ingenieur Informatica, UGent

To become active, a service must be hosted within a run-time environment that creates it and controls its context and lifetime. Windows Communication Foundation (WCF) services are designed to run in any Windows process that supports managed code.

WCF provides a unified programming model for building service-oriented applications. This programming model remains consistent and is independent of the run-time environment in which the service is deployed. In practice, this means that the code for your services looks much the same whatever the hosting option.

These hosting options range from running inside a console application to server environments such as a Windows service running within a worker process managed by Internet Information Services (IIS) or by Windows Process Activation Service (WAS). Developers choose the hosting environment that satisfies the service's deployment requirements. These requirements might derive from the platform on which the application is deployed, the transport on which it must send and receive messages, or on the type of process recycling and other process management required to ensure adequate availability, or on some other management or reliability requirements.

hosting

A service must be hosted in some process. A host is an application that controls the lifetime of the service. Services can be self-hosted or managed by an existing hosting process.

self-hosted service

A service that runs within a process application that the developer created. The developer controls its lifetime, sets the properties of the service, opens the service (which sets it into a listening mode), and closes the service.

hosting process

An application that is designed to host services. These include Internet Information Services (IIS), Windows Activation Services (WAS), and Windows Services. In these hosted scenarios, the host controls the lifetime of the service. For example, using IIS you can set up a virtual directory that contains the service assembly and configuration file. When a message is received, IIS starts the service and controls its lifetime.

instancing

A service has an instancing model. There are three instancing models: "single," in which a single CLR object services all the clients; "per call," in which a new CLR object is created to handle each client call; and "per session," in which a set of CLR objects is created, one for each separate session. The choice of an instancing model depends on the application requirements and the expected usage pattern of the service.

Hosten via console-applicatie

29

```
Uri baseAddress = new Uri("http://localhost:8000/ServiceModelSamples/Service");
using (ServiceHost host = new ServiceHost(typeof(CalculatorService), baseAddress)) {

    host.AddServiceEndpoint( typeof(ICalculator), new WSHttpBinding(), "CalculatorService");
    ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
    smb.HttpGetEnabled = true;
    host.Description.Behaviors.Add(smb);

    host.Open();
    Console.WriteLine("The service is ready at {0}", baseAddress);
    Console.WriteLine("Press <Enter> to stop the service.");
    Console.ReadLine();

    host.Close();
}
```

Industrieel Ingenieur Informatica, UGent

Self-Hosting in a Managed Application

WCF services can be hosted in any managed application. This is the most flexible option because it requires the least infrastructure to deploy. You embed the code for the service inside the managed application code and then create and open an instance of the ServiceHost to make the service available.

This option enables two common scenarios: WCF services running inside console applications and rich client applications such as those based on Windows Presentation Foundation (WPF) or Windows Forms (WinForms). Hosting a WCF service inside a console application is typically useful during the application's development phase. This makes them easy to debug, easy to get trace information from to find out what is happening inside of the application, and easy to move around by copying them to new locations. This hosting option also makes it easy for rich client applications, such as WPF and WinForms applications, to communicate with the outside world. For example, a peer-to-peer collaboration client that uses WPF for its user interface and also hosts a WCF service that allows other clients to connect to it and share information.

To host a service inside a managed application, embed the code for the service inside the managed application code, define an endpoint for the service either imperatively in code, declaratively through configuration, or using default endpoints, and then create an instance of ServiceHost.

To start receiving messages, call Open on ServiceHost. This creates and opens the listener for the service. Hosting a service in this way is often referred to as "self-hosting" because the managed application is doing the hosting work itself. To close the service, call CommunicationObject.Close on ServiceHost.

Hosten als Windowsservice

30

```
public class CalculatorWindowsService : ServiceBase {
    public ServiceHost serviceHost = null;
    public CalculatorWindowsService() {
        ServiceName = "WCFWindowsServiceSample"; }

    public static void Main() {
        ServiceBase.Run(new CalculatorWindowsService()); }

    protected override void OnStart(string[] args) {
        if (serviceHost != null) { serviceHost.Close(); }
        serviceHost = new   ServiceHost(typeof(CalculatorService));
        serviceHost.Open(); }

    protected override void OnStop() {
        if (serviceHost != null) {
            serviceHost.Close();
            serviceHost = null; } } }
```

```
[RunInstaller(true)]
public class ProjectInstaller : Installer {
    private ServiceProcessInstaller process;
    private ServiceInstaller service;
    public ProjectInstaller() {
        process = new ServiceProcessInstaller();
        process.Account = ServiceAccount.LocalSystem;
        service = new ServiceInstaller();
        service.ServiceName =
  "WCFWindowsServiceSample";
        Installers.Add(process);
        Installers.Add(service); }
    }
}
```

Industrieel Ingenieur Informatica, UGent

Managed Windows Services

This hosting option consists of registering the application domain (AppDomain) that hosts an WCF service as a managed Windows Service (formerly known as NT service) so that the process lifetime of the service is controlled by the service control manager (SCM) for Windows services. Like the self-hosting option, this type of hosting environment requires that some hosting code is written as part of the application. The service is implemented as both a Windows Service and as an WCF service by causing it to inherit from the ServiceBase class as well as from an WCF service contract interface. The ServiceHost is then created and opened within an overridden OnStart(String[]) method and closed within an overridden OnStop() method. An installer class that inherits from Installer must also be implemented to allow the program to be installed as a Windows Service by the Installutil.exe tool

The scenario enabled by the managed Windows Service hosting option is that of a long-running WCF service hosted outside of IIS in a secure environment that is not message-activated. The lifetime of the service is controlled instead by the operating system. This hosting option is available in all versions of Windows.

Hosten in IIS

31

- Maak een service in IIS
  - TijdService.svc en TijdService.svc.cs
- Publiceer/Run
- WSDL
  - http://localhost:49695/TijdService.svc?wsdl

Industrieel Ingenieur Informatica, UGent

Internet Information Services (IIS)

The IIS hosting option is integrated with ASP.NET and uses the features these technologies offer, such as process recycling, idle shutdown, process health monitoring, and message-based activation. On the Windows XP and Windows Server 2003 operating systems, this is the preferred solution for hosting Web service applications that must be highly available and highly scalable. IIS also offers the integrated manageability that customers expect from an enterprise-class server product. This hosting option requires that IIS be properly configured, but it does not require that any hosting code be written as part of the application.

Note that IIS-hosted services can only use the HTTP transport.

Windows Process Activation Service (WAS)

Windows Process Activation Service (WAS) is the new process activation mechanism for the Windows Server 2008 that is also available on Windows Vista. It retains the familiar IIS 6.0 process model (application pools and message-based process activation) and hosting features (such as rapid failure protection, health monitoring, and recycling), but it removes the dependency on HTTP from the activation architecture. IIS 7.0 uses WAS to accomplish message-based activation over HTTP. Additional WCF components also plug into WAS to provide message-based activation over the other protocols that WCF supports, such as TCP, MSMQ, and named pipes. This allows applications that use communication protocols to use the IIS features such as process recycling, rapid fail protection, and the common configuration system that were only available to HTTP-based applications.

This hosting option requires that WAS be properly configured, but it does not require you to write any hosting code as part of the application.

Overzicht

- Ontwerp servicecontract
- Implementatie contract
- Configuratie
- Hosting
- Ontwikkelen client

Client ontwikkelen

33

- Ophalen
  - Contract
  - Bindings
  - Adres
- Svcutil.exe

svcutil.exe /language:cs /out:generatedProxy.cs /config:app.config
http://localhost:8000/ServiceModelSamples/service

  - Genereert
    - Configuratiebestand
    - Proxy-klasse
    - Beiden toevoegen aan clientproject
  - Kan ook via WSDL indien gehost op IIS

Industrieel Ingenieur Informatica, UGent

A client application is a managed application that uses a WCF client to communicate with another application. To create a client application for a WCF service requires the following steps:

Obtain the service contract, bindings, and address information for a service endpoint.
Create a WCF client using that information.
Call operations.
Close the WCF client object.

In WCF, services and clients model contracts using managed attributes, interfaces, and methods. To connect to a service in a client application, you need to obtain the type information for the service contract. Typically, you do this by using the ServiceModel Metadata Utility Tool (Svcutil.exe), which downloads metadata from the service, converts it to a managed source code file in the language of your choice, and creates a client application configuration file that you can use to configure your WCF client object.

client application

A program that exchanges messages with one or more endpoints. The client application begins by creating an instance of a WCF client and calling methods of the WCF client. It is important to note that a single application can be both a client and a service.

metadata

In a service, describes the characteristics of the service that an external entity needs to understand to communicate with the service. Metadata can be consumed by the ServiceModel Metadata Utility Tool (Svcutil.exe) to generate a WCF client and accompanying configuration that a client application can use to interact with the service.

The metadata exposed by the service includes XML schema documents, which define the data contract of the service, and WSDL documents, which describe the methods of the service.

When enabled, metadata for the service is automatically generated by WCF by inspecting the service and its endpoints. To publish metadata from a service, you must explicitly enable the metadata behavior.

WCF client

A client-application construct that exposes the service operations as methods (in the .NET Framework programming language of your choice, such as Visual Basic or Visual C#). Any application can host a WCF client, including an application that hosts a service. Therefore, it is possible to create a service that includes WCF clients of other services.

A WCF client can be automatically generated by using the ServiceModel Metadata Utility Tool (Svcutil.exe) and pointing it at a running service that publishes metadata.

Proxy gebruiken

34

- Proxy aanmaken
- Methodes proxy oproepen
- Proxy afsluiten

Industrieel Ingenieur Informatica, UGent

A WCF client is a local object that represents a WCF service in a form that the client can use to communicate with the remote service. WCF client types implement the target service contract, so when you create one and configure it, you can then use the client object directly to invoke service operations. The WCF run time converts the method calls into messages, sends them to the service, listens for the reply, and returns those values to the WCF client object as return values or out or ref parameters.

Proxy gebruiken

35

```
class Client {
   static void Main() {
         CalculatorClient client = new CalculatorClient();
         double value1 = 100.00D;
         double value2 = 15.99D;
         double result = client.Add(value1, value2);

         client.Close();
   }
}
```

Voorbeeld REST

36

- ComponistService.svc
- http://localhost:50633/ComponistService.svc/GetComponisten?letters=b


```
{"d":[{"__type":"Componist:#Componisten","Category":"Baroque","F
irstName":"Johann Sebastian","Id":"1","LastName":"Bach"},
{"__type":"Componist:#Componisten","Category":"Classical","First
Name":"Ludwig van","Id":"8","LastName":"Beethoven"},
{"__type":"Componist:#Componisten","Category":"Classical","First
Name":"Johannes","Id":"9","LastName":"Brahms"},
{"__type":"Componist:#Componisten","Category":"Romantic","FirstN
ame":"Louis-Hector","Id":"20","LastName":"Berlioz"},
{"__type":"Componist:#Componisten","Category":"Romantic","FirstN
ame":"Georges","Id":"21","LastName":"Bizet"},
{"__type":"Componist:#Componisten","Category":"Romantic","FirstN
ame":"Bedrich","Id":"37","LastName":"Smetana"},
{"__type":"Componist:#Componisten","Category":"Post-
Romantic","FirstName":"Bela","Id":"41","LastName":"Bartok"},
{"__type":"Componist:#Componisten","Category":"Post-
Romantic","FirstName":"Leonard","Id":"42","LastName":"Bernstein"
},{"__type":"Componist:#Componisten","Category":"Post-
Romantic","FirstName":"Benjamin","Id":"43","LastName":"Britten"}
]}
```

Informatie

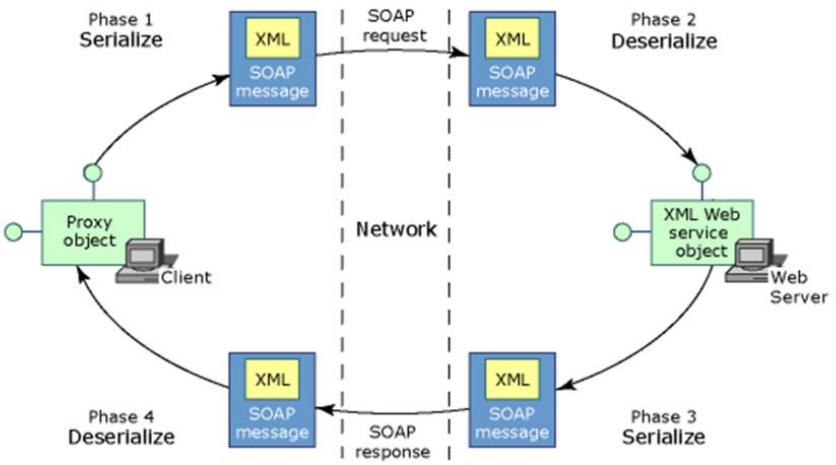
- WCF (http://msdn.microsoft.com/en-us/library/dd456779.aspx)

WEBSERVICES IN JAVA

Veerle Ongenae

Overzicht

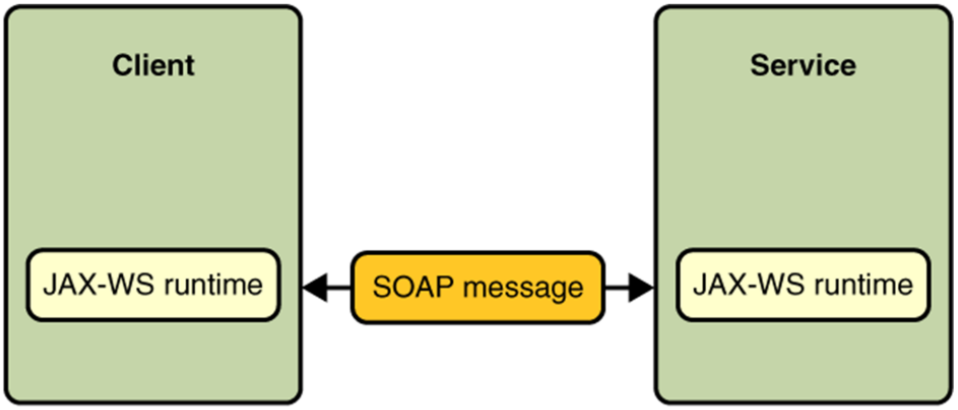
2



Industrieel Ingenieur Informatica, Universiteit Gent

JAX-WS Principe

3



Industrieel Ingenieur Informatica, Universiteit Gent

Java API for XML Web Services (JAX-WS) is a technology for building web services and clients that communicate using XML. JAX-WS allows developers to write message-oriented as well as Remote Procedure Call–oriented (RPC-oriented) web services.

In JAX-WS, a web service operation invocation is represented by an XML-based protocol, such as SOAP. The SOAP specification defines the envelope structure, encoding rules, and conventions for representing web service invocations and responses. These calls and responses are transmitted as SOAP messages (XML files) over HTTP.

Although SOAP messages are complex, the JAX-WS API hides this complexity from the application developer. On the server side, the developer specifies the web service operations by defining methods in an interface written in the Java programming language. The developer also codes one or more classes that implement those methods. Client programs are also easy to code. A client creates a proxy (a local object representing the service) and then simply invokes methods on the proxy. With JAX-WS, the developer does not generate or parse SOAP messages. It is the JAX-WS runtime system that converts the API calls and responses to and from SOAP messages.

With JAX-WS, clients and web services have a big advantage: the platform independence of the Java programming language. In addition, JAX-WS is not restrictive: A JAX-WS client can access a web service that is not running on the Java platform and vice versa. This flexibility is possible because JAX-WS uses technologies defined by the W3C: HTTP, SOAP, and WSDL. WSDL specifies an XML format for describing a service as a set of endpoints operating on messages.

Voorbeeld

4

- Boek-object opvragen aan de hand van ISBN

Web Services

Endpoint	Information
Service Name: {http://web.boeken.iii.be/}Catalogus Port Name: {http://web.boeken.iii.be/}CatalogusPort	Address: http://localhost:8080/WebCatalogus/Catalogus WSDL: http://localhost:8080/WebCatalogus/Catalogus?wsdl Implementation class: be.iii.boeken.web.Catalogus

Catalogus Web Service Tester

This form will allow you to test your web service implementation (WSDL File)

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

Methods :

public abstract be.iii.boeken.web.Boek be.iii.boeken.web.Catalogus.geefBoek(java.lang.String)
geefBoek (isbn2)

Industrieel Ingenieur Informatica, Universiteit Gent

SOAP-aanvraag

5

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope
   xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:geefBoek xmlns:ns2="http://web.boeken.iii.be/">
      <isbn>isbn2</isbn>
    </ns2:geefBoek>
  </S:Body>
</S:Envelope>
```

Industrieel Ingenieur Informatica, Universiteit Gent

SOAP 1.1

SOAP-antwoord

6

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:geefBoekResponse xmlns:ns2="http://web.boeken.iii.be/">
      <return>
         <isbn>isbn2</isbn>
         <prijs>36.0</prijs>
         <titel>Java Servlet Programming</titel>
      </return>
    </ns2:geefBoekResponse>
  </S:Body>
</S:Envelope>
```

Industrieel Ingenieur Informatica, Universiteit Gent

WSDL

7

- WSDL: Webservice Description Language
- http://localhost:8080/WebCatalogus/Catalogus?WSDL
- http://localhost:8080/WebCatalogus/Catalogus?xsd=1

Industrieel Ingenieur Informatica, Universiteit Gent

WSDL 1.1

Informatie over webservice → genereren proxy-object

De structuur van de SOAP-berichten

Methodes

Argumenten

Teruggeefwaarden

Locatie webservices

Transportprotocol

Werkwijze in J2EE

8

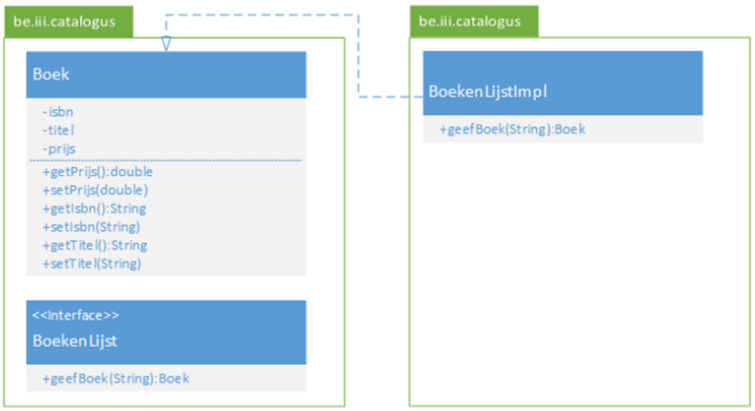
- Maken webservice
  - Klassen, interface, … met eigenlijke functionaliteit maken (bo en jdbc)
  - Webservice maken
  - Webservice publiceren
- Clientapplicatie
  - Op basis van WSDL proxy-klassen genereren (wsimport.exe)
  - Clientapplicatie maakt gebruik van proxy-klassen/objecten

Industrieel Ingenieur Informatica, Universiteit Gent

J2EE = Java 2 Entreprise Edition

Eigenlijke functionaliteit

9



Industrieel Ingenieur Informatica, Universiteit Gent

be.iii.catalogus.Boek

Moet geserialiseerd worden

Default-constructor

get/set-methodes voor id, titel en prijs

anders NIET in XML

“eenvoudige” types

be.iii.catalogus.BoekenLijst

Interface

Methode geefBoek

be.iii.catalogus.impl.BoekenLijstImpl

Implementatie interface BoekenLijst

In jar catalogus.jar

Webservice

10

```
@WebService(serviceName = "Catalogus")
public class Catalogus {
    @WebMethod(operationName = "geefBoek")
    public Boek geefBoek(@WebParam(name = "isbn") String isbn) {
        try {
            BoekenLijst boeken = new BoekenLijstImpl();
            return boeken.geefBoek(isbn);
        } catch (Exception e) {
            return null;
        }
    }
}
```

Industrieel Ingenieur Informatica, Universiteit Gent

Klasse met annotatie @Webservice()

Methode met annotatie @WebMethod

Parameters met annotatie @WebParam(…)

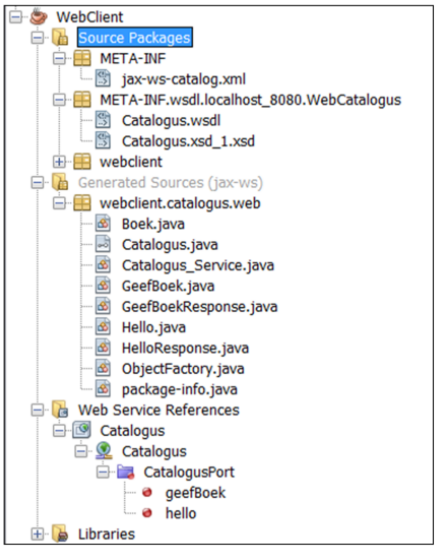
Webservice publiceren

11

- Webservice deployen
  - War-bestand
  - Publiceren webservice
    - Wordt gewrapt in een servlet
- WSDL automatisch gegenereerd
  - Wsdl af te halen via http://localhost:8080/WebCatalogus/Catalogus?wsdl
  - wsgen: tool om dit handmatig te doen

Genereren proxy-klassen

12



Industrieel Ingenieur Informatica, Universiteit Gent

Automatisch in Netbeans in een project

New → Web Service Client

Url van de WSDL doorgeven

Gegeneerd

Package META-INF

jax-ws-catalog.xml

Locatie WSDL

META-INF.wsdl.localhost_8080.WebCatalogus

Catalogus.wsdl

Catalogus.xsd_1.xsd

Klassen voor proxy

webclient.catalogus.web

In Projectsvenster

Map Web Service References

Catalogus

WebClient


```
public static void main(String[] args) {
    for (int i = 2; i <= 3; i++) {
        String isbn = Integer.toString(i);
        Boek boek = geefBoek("isbn"+isbn);
        System.out.println(boek.getTitel());
    }
}
private static Boek geefBoek(String isbn) {
    webclient.catalogus.web.Catalogus_Service service
            = new webclient.catalogus.web.Catalogus_Service();
    webclient.catalogus.web.Catalogus port
            = service.getCatalogusPort();
    return port.geefBoek(isbn);
}
```


Project WebClient

webclient.WebClient.java

Oproepen methode webservice

Rechts klikken in editeervenster

Insert Code

Call Web Service Operation

Genereert de nodige code om de proxyobjecten te gebruiken

Informatie over webservices

14

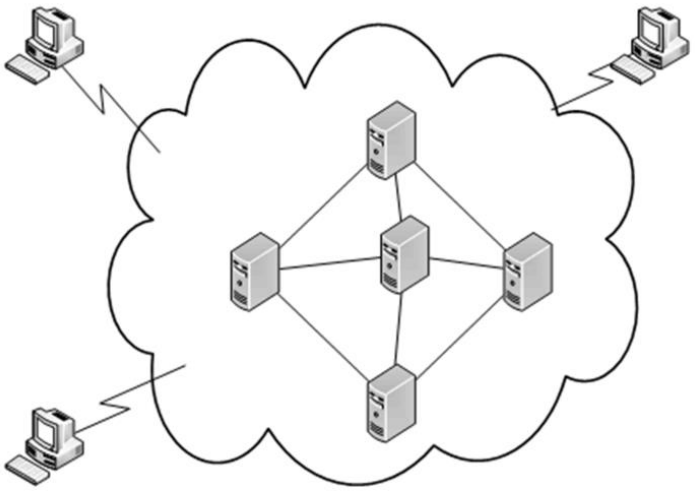
- The JavaEE 7 Tutorial Part VI Web Services (https://docs.oracle.com/javaee/7/tutorial/partwebsvcs.htm#BNAYK)
- Webservices in Netbeans (http://www.netbeans.org/kb/trails/web.html)

WEBSERVICES

Veerle Ongenae

Gedistribueerde applicaties

2



https://msdn.microsoft.com/en-us/library/cc239737.aspx

Industrieel Ingenieur Informatica, Universiteit Gent

Gedistribueerde applicaties/Gedistribueerd systeem
- Verschillende stukken software
  - Op computers in het netwerk
- Samenwerken
  - Via berichten
- Taken uitvoeren
  - Lijkt één applicatie voor de gebruiker

Verschillende spelers
- Verschillende computers
- Verschillende omgevingen

Gevolgen

Geen ruimtelijke beperking
- Moeilijk beheerbaar?

Aanvraag faalt. Oorzaak?
- Een van de softwarecomponenten?
- Netwerk?

Geen globale tijd

- Lokaal gegenereerde timestamps niet bruikbaar
- Geen globale toestand/status
- Softwarecomponenten kunnen parallel uitgevoerd worden
  - Inconsistenties mogelijk

Voordelen

- Bronnen delen
  - Informatie
  - Hardware
- Schaalbaar
- Kan makkelijker fouten opvangen

Gedistribueerde applicaties

3

- EDI

EDI

4

http://www.scaneurope.nl/wp/?page_id=396

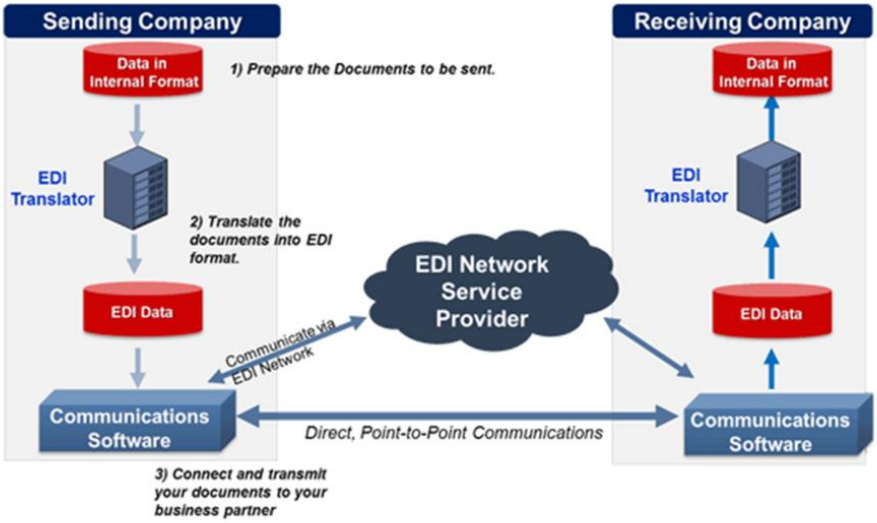
Industrieel Ingenieur Informatica, Universiteit Gent

Electronic Data Interchange (EDI) may be most easily understood as the replacement of paper-based purchase orders with electronic equivalents. It is actually much broader in its application than the procurement process, and its impacts are far greater than mere automation. EDI offers the prospect of easy and cheap communication of structured information throughout the corporate community, and is capable of facilitating much closer integration among hitherto remote organisations.

A more careful definition of EDI is 'the exchange of documents in standardised electronic form, between organisations, in an automated manner, directly from a computer application in one organisation to an application in another'.

EDI - architectuur

5



http://www.edibasics.com/what-is-edi/how-does-edi-work/

Industrieel Ingenieur Informatica, Universiteit Gent

EDI can be compared and contrasted with electronic mail (email). Email enables free-format, textual messages to be electronically transmitted from one person to another. EDI, on the other hand, supports structured business messages (those which are expressed in hard-copy, pre-printed forms or business documents), and transmits them electronically between computer applications, rather than between people.

The essential elements of EDI are:

the use of **an electronic transmission medium** (originally a value-added network, but increasingly the open, public Internet) rather than the despatch of physical storage media such as magnetic tapes and disks;

the use of **structured, formatted messages based on agreed standards** (such that messages can be translated, interpreted and checked for compliance with an explicit set of rules);

relatively fast delivery of electronic documents from sender to receiver (generally implying receipt within hours, or even minutes); and

direct communication between applications (rather than merely between computers).

EDI depends on a moderately sophisticated information technology infrastructure. This must include data processing, data management and networking capabilities, to enable the efficient capture of data into electronic form, the processing and retention of data, controlled access to it, and efficient and reliable data transmission between remote sites.

A common connection point is needed for all participants, together with a set of electronic mailboxes (so that the organisations' computers are not interrupted by one another), and security and communications management features. It is entirely feasible for organisations to implement EDI directly with one another, but it generally proves advantageous to use a third-party network services provider.

Advantages of EDI

EDI provides cost savings by reducing paper and eliminating paper processing.

Time savings and eliminating repetition are other benefits from the reduction in paper processing.

Documents can be transferred more quickly and processing errors can be decreased allowing business to be done more efficiently.

More efficient processing will likely lead to improved customer service which will ultimately expand the customer base.

Disadvantages of EDI

Contrasted to XML, which is not strictly standardized, many consider EDI to have too many standards.

There are various standards bodies who have developed 'standard document formats' for EDI which can cause problems with cross compatibility.

These standards bodies also push standards revisions annually which could cause problems if you have a more recent version of a document than a business partner.

EDI systems are extremely expensive making it difficult for small businesses to implement.

Many large organizations will only work with others who utilize EDI. This may limit the business small companies are able to do with such organizations and limit trading partners.

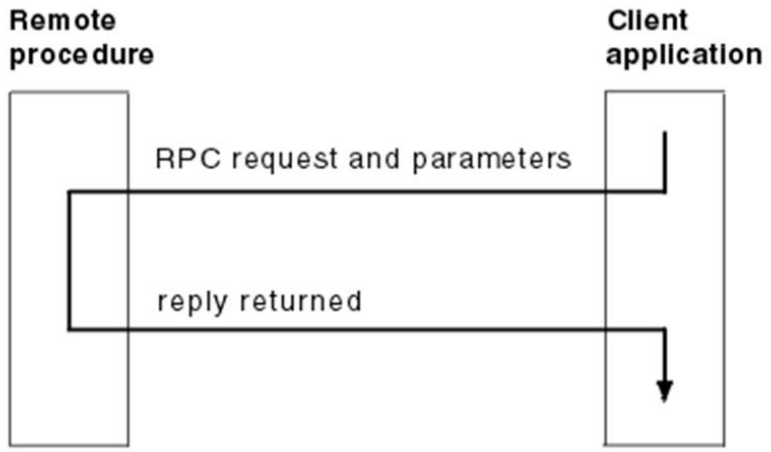
Gedistribueerde applicaties

6

- EDI
- RPC

RPC

7



http://www-01.ibm.com/support/knowledgecenter/SSGMCP_4.1.0/com.ibm.cics.ts.doc/dfhtm/topics/dfhtmc00188.html

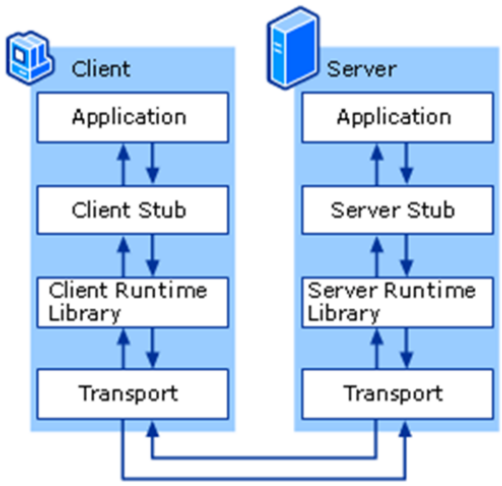
Industrieel Ingenieur Informatica, Universiteit Gent

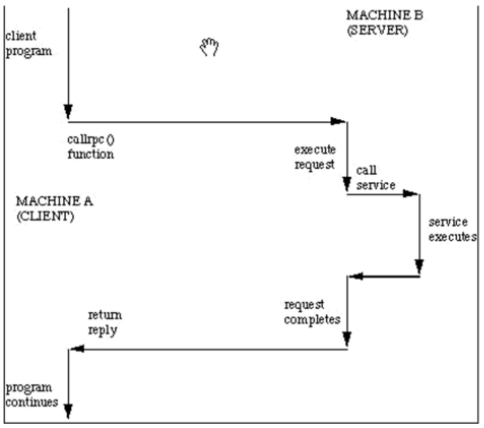
When a process invokes or calls a process on a remote system, that call is a *remote procedure call (RPC)*.

RPC is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using RPC, programmers of distributed applications avoid the details of the interface with the network. The transport independence of RPC isolates the application from the physical and logical elements of the data communications mechanism and allows the application to use a variety of transports.

Werking RPC

8





http://www.cs.cf.ac.uk/Dave/C/node33.html#SECTION00331000000000000000

https://technet.microsoft.com/en-us/library/cc738291%28v=ws.10%29.aspx

Industrieel Ingenieur Informatica, Universiteit Gent

The RPC process starts on the client side. The client application calls a local stub procedure instead of code implementing the procedure. Stubs are compiled and linked with the client application during development. Instead of containing code that implements the remote procedure, the client stub code retrieves the required parameters from the client address space and delivers them to the client runtime library. The client runtime library then translates the parameters as needed into a standard Network Data Representation (NDR) format for transmission to the server.

When the server receives the RPC, either locally or from a remote client, the server RPC runtime library functions accept the request and call the server stub procedure. The server stub retrieves the parameters from the network buffer and, using one of the NDR marshalling engines, converts them from the network transmission format to the format required by the server. The server stub calls the actual procedure on the server. The remote procedure then runs, possibly generating output parameters and a return value. When the remote procedure is complete, a similar sequence of steps returns the data to the client.

The remote procedure returns its data to the server stub which, using one of the NDR marshalling engines, converts output parameters to the format required for transmission back to the client and returns them to the RPC

runtime library functions. The server RPC runtime library functions transmit the data to the client computer using either LRPC or the network.

The client completes the process by accepting the data over the network and returning it to the calling function. The client RPC runtime library receives the remote-procedure return values, converts the data from its NDR to the format used by the client computer, and returns them to the client stub.

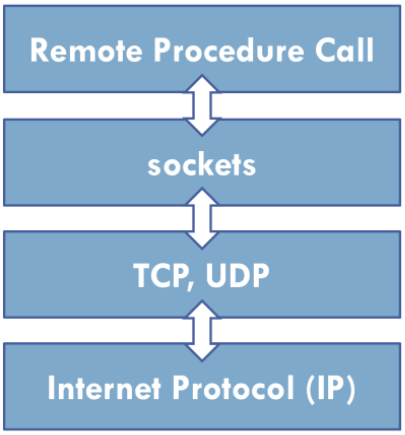
The server application contains calls to the server runtime library functions, which register the server's interface with the RPC runtime and allow the server to accept remote procedure calls. The server application also contains the application-specific remote procedures that are called by the client applications.

The client stub then calls functions in the RPC client runtime library to send the request and its parameters to the server. If the server is located on the same host as the client, the runtime library can use the Local RPC (LRPC) function and pass the RPC request to the Windows kernel for transport to the server. If the server is located on a remote host, the runtime library specifies an appropriate transport protocol engine and passes the RPC to the network stack for transport to the server.

An RPC is analogous to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. The second figure shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

Vaak in C.

Communicatie tussen applicaties



Remote Procedure Call:
Verbergt communicatie details als een aanroep van een procedure op een server ~ proxy pattern

sockets:
API voor onderliggende communicatieprotocollen

TCP, UDP:
UDP: transport datapakketen zonder garantie
TCP: verzekerde datastromen

Internet Protocol (IP):
Verstuurt datapakketten

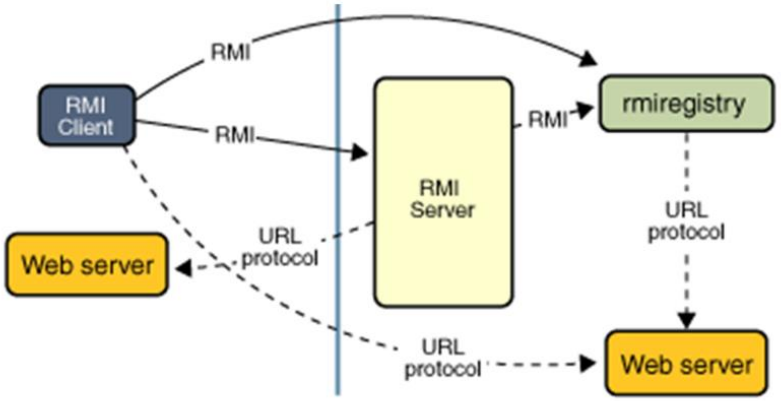
Gedistribueerde applicaties

10

- EDI
- RPC
- RMI, DCOM, CORBA

RMI

11



https://docs.oracle.com/javase/tutorial/rmi/overview.html

Industrieel Ingenieur Informatica, Universiteit Gent

Remote Method Invocation - Java

RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.

Distributed object applications need to do the following:

Locate remote objects. Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.

Communicate with remote objects. Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.

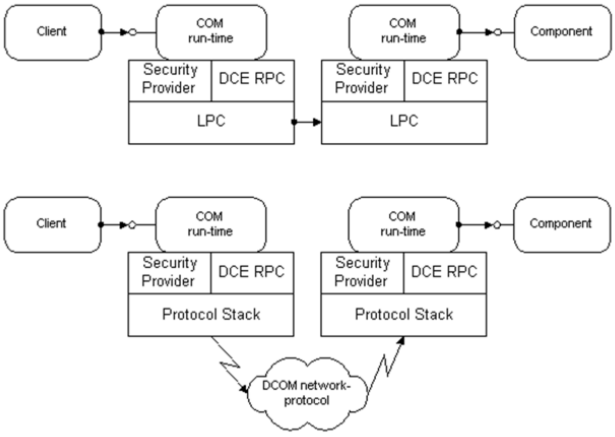
Load class definitions for objects that are passed around. Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's

data.

The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.

DCOM

12



http://www.active-undelete.com/dcom-overview.htm

The Distributed Component Object Model (DCOM) Remote Protocol is a protocol for exposing application objects by way of remote procedure calls (RPCs). The protocol consists of a set of extensions layered on Microsoft Remote Procedure Call Protocol Extensions as specified in [MS-RPCE].

The DCOM Remote Protocol is also referred to as Object RPC or ORPC.

Microsoft® Distributed COM (DCOM) extends the Component Object Model (COM) to support communication among objects on different computers-on a LAN, a WAN, or even the Internet. Because DCOM is a seamless evolution of COM you can take advantage of your existing investment in COM-based applications, components, tools, and knowledge to move into the world of standards-based distributed computing. DCOM handles low-level details of network protocols so you can focus on providing solutions to your customers.

DCOM is an extension of the Component Object Model (COM). COM defines how components and their clients interact. This interaction is defined such that the client and the component can connect without the need of any intermediary system component. The client calls methods in the component without any overhead whatsoever.

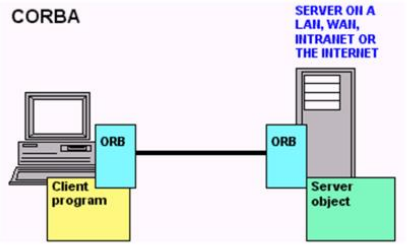
In today's operating systems, processes are shielded from each other. A client that needs to communicate with a component in another process cannot call the component directly, but has to use some form of interprocess communication provided by the operating system. COM provides this communication in a completely transparent fashion: it intercepts calls from the client and forwards them to the component in another process. The first figure illustrates how the COM/DCOM run-time libraries provide the link between client and component.

When client and component reside on different machines, DCOM simply replaces the local interprocess communication with a network protocol. Neither the client nor the component is aware that the wire that connects them has just become a little longer.

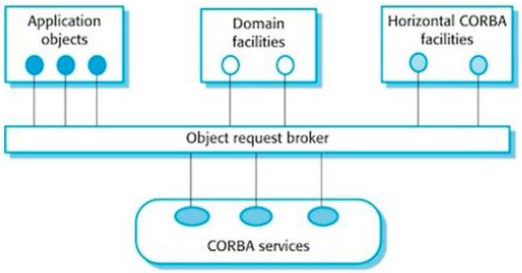
The second figure shows the overall DCOM architecture: The COM run-time provides object-oriented services to clients and components and uses RPC and the security provider to generate standard network packets that conform to the DCOM wire-protocol standard.

CORBA

13



http://www.pcmag.com/encyclopedia/term/40365/corba



https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/DistribSys/Corba.html

Industrieel Ingenieur Informatica, Universiteit Gent

CORBA, or Common Object Request Broker Architecture, is a standard architecture for distributed object systems. It allows a distributed, heterogeneous collection of objects to interoperate.

The Object Management Group (OMG) is responsible for defining CORBA. The OMG comprises over 700 companies and organizations, including almost all the major vendors and developers of distributed object technology, including platform, database, and application vendors as well as software tool and corporate developers.

CORBA defines an architecture for distributed objects. The basic CORBA paradigm is that of a request for services of a distributed object. Everything else defined by the OMG is in terms of this basic paradigm.

The services that an object provides are given by its interface. Interfaces are defined in OMG's Interface Definition Language (IDL). Distributed objects are identified by object references, which are typed by IDL interfaces.

Application objects that are designed and implemented for this application.

Standard objects that are defined by the OMG for a specific domain. These domain object standards cover finance/insurance, electronic commerce, healthcare, and a number of other areas.

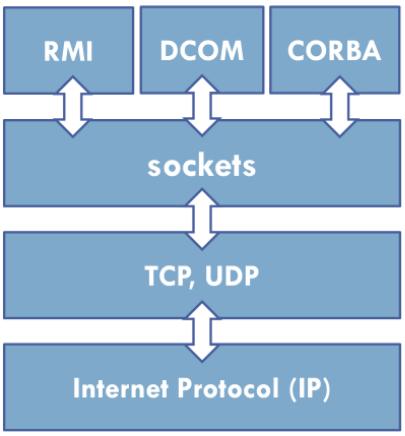
Fundamental CORBA services that provide basic distributed computing services such as directories, security management, etc.

Horizontal CORBA facilities such as user interface facilities, system management facilities, etc. The term horizontal facilities suggests that these facilities are common to many application domains and the facilities are therefore used in many different applications.

Communicatie tussen applicaties

14

- Gedistribueerd programmeren populairder → nood aan extra functionaliteit en flexibiliteit



Gespecialiseerde middleware, volgens RPC-model, met extra functionaliteit

sockets:
API voor onderliggende communicatieprotocollen

TCP, UDP:
UDP: transport datapakketen zonder garantie
TCP: verzekerde datastromen

Internet Protocol (IP):
Verstuurt datapakketten

Industrieel Ingenieur Informatica, Universiteit Gent

Gedistribueerde object-infrastructuur over het Internet

Transportprotocol bovenop TCP/IP (verschillend voor CORBA, RMI en DCOM) voor verschillende platformen

Afspraken over communicatie tussen objecten

Voordeel: de verschillende servers hoeven niet tot het zelfde netwerk te behoren

Nadeel: Kunnen enkel met dezelfde infrastructuur praten, tenzij extra lagen op een reeds ingewikkelde structuur

Gedistribueerde applicaties

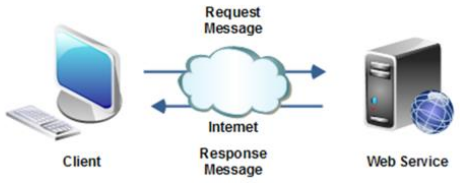
15

- EDI
- RPC
- RMI, DCOM, CORBA
- Webservices
  - Definitie

Webservice - definitie

16

- Software service (dienst)
- Toegankelijk via internet
  - Via bestaande protocollen bv. HTTP



http://tutorials.jenkov.com/web-services/message-formats.html

Industrieel Ingenieur Informatica, Universiteit Gent

Softwaresysteem

Communicatie tussen computers (applicaties)

Over een netwerk

Mogelijkheid tot het uitwisselen van data overal op het web

Geen koppeling tussen data en transport

Meestal niet binair

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Services versus Applications

Services

Perform a single or a few specialized operations.

Most often accessed by other programs.

Often (but not always) targets part of a larger problem domain.

Applications

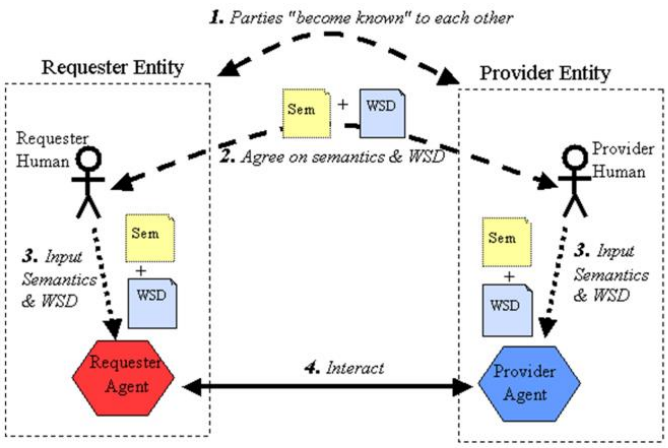
Perform a wide range of operations, and may even expose some of these operations as services.

Often (but not always) accessed by humans.

Often (but not always) targets a whole problem domain.

Webservice – entities

17



http://www.w3.org/TR/ws-arch/

Industrieel Ingenieur Informatica, Universiteit Gent

Interface

- Beschreven in een formaat leesbaar door computers
- WSDL

Interactie met webservice

- SOAP-berichten
  - Meestal over HTTP
  - Inhoud XML

A Web service is an abstract notion that must be implemented by a concrete agent. The agent is the concrete piece of software or hardware that sends and receives messages, while the service is the resource characterized by the abstract set of functionality that is provided. To illustrate this distinction, you might implement a particular Web service using one agent one day (perhaps written in one programming language), and a different agent the next day (perhaps written in a different programming language) with the same functionality. Although the agent may have changed, the Web service remains the same.

The purpose of a Web service is to provide some functionality on behalf of its owner -- a person or organization, such as a business or an individual.

The provider entity is the person or organization that provides an appropriate agent to implement a particular service.

A requester entity is a person or organization that wishes to make use of a provider entity's Web service. It will use a requester agent to exchange messages with the provider entity's provider agent.

(In most cases, the requester agent is the one to initiate this message exchange, though not always. Nonetheless, for consistency we still use the term "requester agent" for the agent that interacts with the provider agent, even in cases when the provider agent actually initiates the exchange.)

In order for this message exchange to be successful, the requester entity and the provider entity must first agree on both the semantics and the mechanics of the message exchange.

The mechanics of the message exchange are documented in a Web service description (WSD). The WSD is a machine-processable specification of the Web service's interface, written in WSDL. It defines the message formats, datatypes, transport protocols, and transport serialization formats that should be used between the requester agent and the provider agent. It also specifies one or more network locations at which a provider agent can be invoked, and may provide some information about the message exchange pattern that is expected. In essence, the service description represents an agreement governing the mechanics of interacting with that service.

The semantics of a Web service is the shared expectation about the behavior of the service, in particular in response to messages that are sent to it. In effect, this is the "contract" between the requester entity and the provider entity regarding the purpose and consequences of the interaction. Although this contract represents the overall agreement between the requester entity and the provider entity on how and why their respective agents will interact, it is not necessarily written or explicitly negotiated. It may be explicit or implicit, oral or written, machine processable or human oriented, and it may be a legal agreement or an informal (non-legal) agreement.

While the service description represents a contract governing the mechanics of interacting with a particular service, the semantics represents a contract governing the meaning and purpose of that interaction. The dividing line between these two is not necessarily rigid. As more semantically rich languages are used to describe the mechanics of the interaction, more of the essential information may migrate from the informal semantics to the service description. As this migration occurs, more of the work required to achieve successful interaction can be automated.

Webservice – andere terminologie

18

Industrieel Ingenieur Informatica, Universiteit Gent

Service provider

Biedt een webservice aan

Service requester

Maakt gebruik van een webservice

A word on terminology: Many documents use the term service provider to refer to the provider entity and/or provider agent. Similarly, they may use the term service requester to refer to the requester entity and/or requester agent. However, since these terms are ambiguous -- sometimes referring to the agent and sometimes to the person or organization that owns the agent -- this document prefers the terms requester entity, provider entity, requester agent and provider agent.

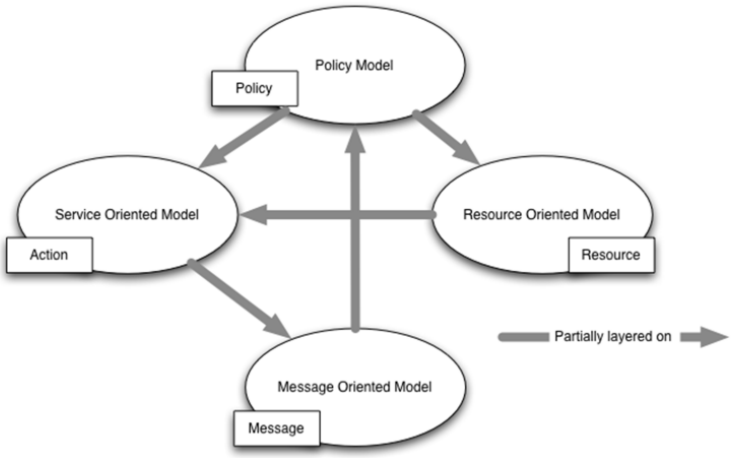
Gedistribueerde applicaties

19

- EDI
- RPC
- RMI, DCOM, CORBA
- Webservices
  - Definitie
  - Architectuur

Architecturale modellen

20



http://www.w3.org/TR/ws-arch/

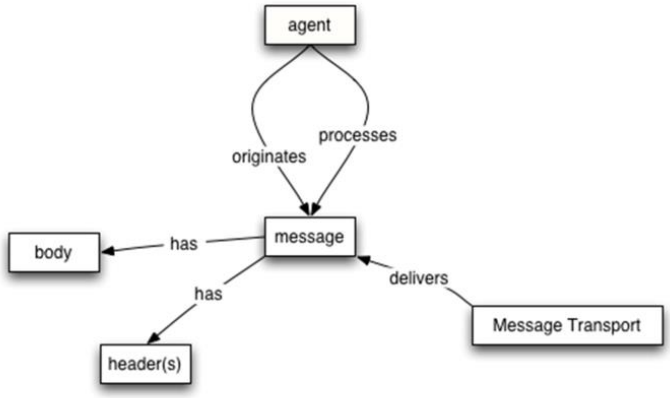
Industrieel Ingenieur Informatica, Universiteit Gent

A model is a coherent subset of the architecture that typically revolves around a particular aspect of the overall architecture. Although different models share concepts, it is usually from different points of view; the major role of a model is to explain and encapsulate a significant theme within the overall Web services architecture.

For example, the Message Oriented Model focuses and explains Web services strictly from a message passing perspective. In particular, it does not attempt to relate messages to services provided. The Service Oriented Model, however, lays on top of and extends the Message Oriented Model in order to explain the fundamental concepts involved in service - in effect to explain the purpose of the messages in the Message Oriented Model.

Message Oriented Model

21



http://www.w3.org/TR/ws-arch/

Industrieel Ingenieur Informatica, Universiteit Gent

Simplified Message Oriented Model

The Message Oriented Model focuses on messages, message structure, message transport and so on — without particular reference as to the reasons for the messages, nor to their significance.

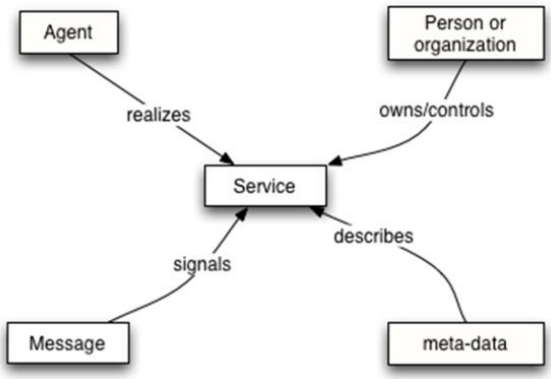
The essence of the message model revolves around a few key concepts illustrated above: the agent that sends and receives messages, the structure of the message in terms of message headers and bodies and the mechanisms used to deliver messages. Of course, there are additional details to consider: the role of policies and how they govern the message level model.

A message represents the data structure passed from its sender to its recipients. The structure of a message is defined in a service description.

Examples of delivery policies include quality of service assurances — such as reliable versus best effort message delivery — and security assurances — such as encrypted versus unencrypted message transport. Another kind of delivery policy could take the form of assertions about recording an audit of how the message was delivered.

Service Oriented Model

22



http://www.w3.org/TR/ws-arch/

Industrieel Ingenieur Informatica, Universiteit Gent

Simplified Service Oriented Model

The Service Oriented Model focuses on aspects of service, action and so on. While clearly, in any distributed system, services cannot be adequately realized without some means of messaging, the converse is not the case: messages do not need to relate to services.

The Service Oriented Model is the most complex of all the models in the architecture. However, it too revolves around a few key ideas. A service is realized by an agent and used by another agent. Services are mediated by means of the messages exchanged between requester agents and provider agents.

A very important aspect of services is their relationship to the real world: services are mostly deployed to offer functionality in the real world. We model this by elaborating on the concept of a service's owner — which, whether it is a person or an organization, has a real world responsibility for the service.

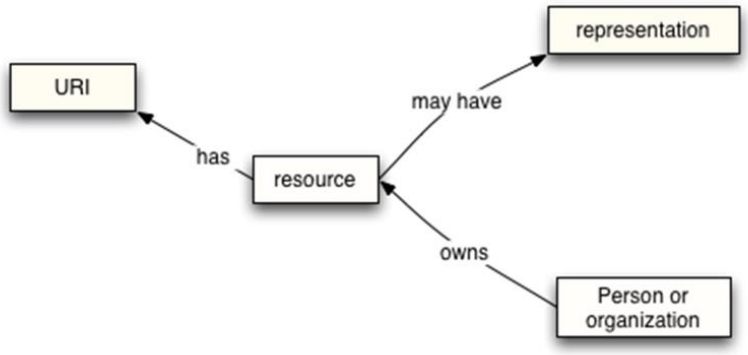
Finally, the Service Oriented Model makes use of meta-data, which is a key property of Service Oriented Architectures. This meta-data is used to

document many aspects of services: from the details of the interface and transport binding to the semantics of the service and what policy restrictions there may be on the service. Providing rich descriptions is key to successful deployment and use of services across the Internet.

A service is an abstract resource that represents a person or organization in some collection of related tasks as having specific service roles. The service may be realized by one or more provider agents that act on behalf of the person or organization — the provider entity.

Resource Oriented Model

23



http://www.w3.org/TR/ws-arch/

Industrieel Ingenieur Informatica, Universiteit Gent

Simplified Resource Oriented Model

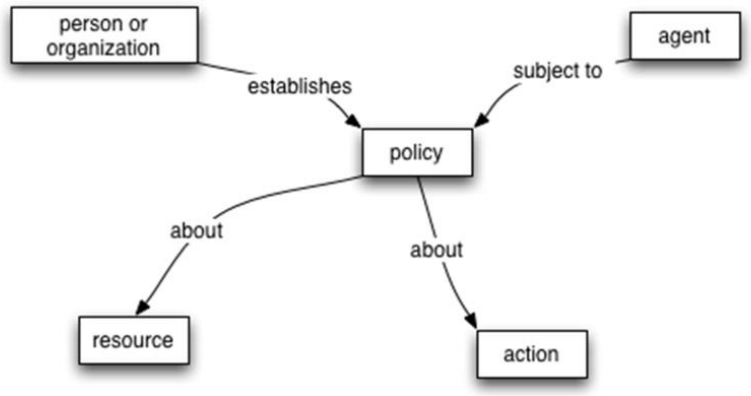
The Resource Oriented Model focuses on resources that exist and have owners.

The resource model is adopted from the Web Architecture concept of resource. We expand on this to incorporate the relationships between resources and owners.

Discovery is the act of locating a machine-processable description of a Web service-related resource that may have been previously unknown and that meets certain functional criteria. It involves matching a set of functional and other criteria with a set of resource descriptions. The goal is to find an appropriate Web service-related resource.

Policy Model

24



http://www.w3.org/TR/ws-arch/

Industrieel Ingenieur Informatica, Universiteit Gent

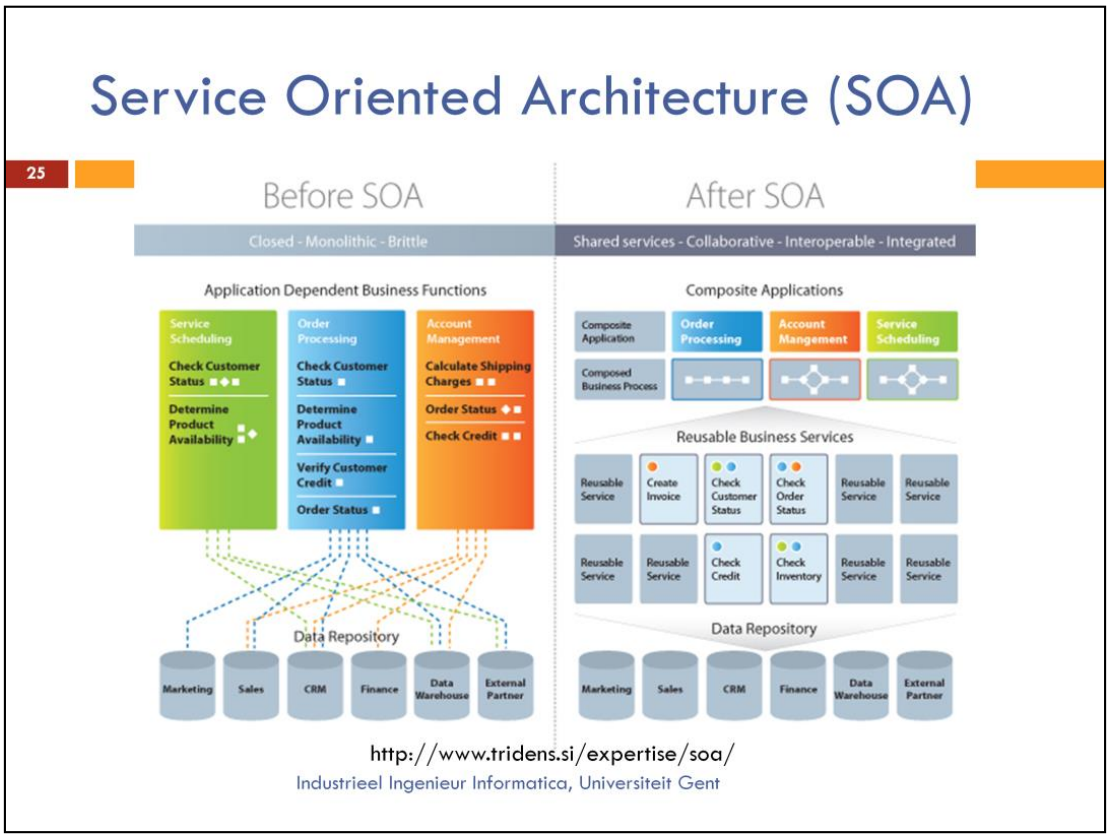
Simplified Policy Model

The Policy Model focuses on constraints on the behavior of agents and services. We generalize this to resources since policies can apply equally to documents (such as descriptions of services) as well as active computational resources.

Policies are about resources. They are applied to agents that may attempt to access those resources, and are put in place, or established, by people who have responsibility for the resource.

Policies may be enacted to represent security concerns, quality of service concerns, management concerns and application concerns.

Beperkingen en verplichtingen.



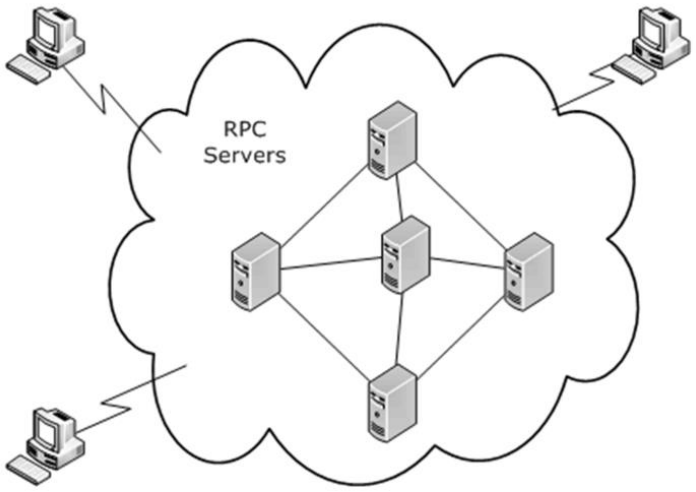
SOA is more than an IT solution; it is a paradigm shift, taking the focus off IT and putting it back where it belongs – on customers, service quality, and fundamental business objectives. The overall benefits are a greater return on investment (ROI) and a more flexible technology system, one better able to support the implementation of your strategic decisions.

Basically, SOA is about describing, building, using, and managing your IT environment, concentrating upon the services it supplies, rather than upon the technology it employs. When enterprises adopt a service-oriented architecture, they describe their IT investment in terms of modular services, each delivering specific value. A well architected, governed, and managed SOA environment enables enterprises to quickly create, combine, and deploy services so as to respond to rapidly changing business needs.

Architectuur voor gedistribueerde systemen

Gedistribueerde systemen

26



https://msdn.microsoft.com/en-us/library/cc239737.aspx

Industrieel Ingenieur Informatica, Universiteit Gent

A **distributed system** consists of diverse, discrete software agents that must work together to perform some tasks. Furthermore, the agents in a distributed system do not operate in the same processing environment, so they must communicate by hardware/software protocol stacks over a network. This means that communications with a distributed system are **intrinsically less fast and reliable** than those using direct code invocation and shared memory. This has important architectural implications because distributed systems require that developers (of infrastructure and applications) **consider the unpredictable latency of remote access**, and take into account **issues of concurrency and the possibility of partial failure**.

Distributed object systems are distributed systems in which the semantics of object initialization and method invocation are exposed to remote systems by means of a proprietary or standardized mechanism to broker requests across system boundaries, marshall and unmarshall method argument data, etc. Distributed objects systems typically (albeit not necessarily) are characterized by objects maintaining a fairly complex internal state required to support their methods, a fine grained or "chatty" interaction between an object and a program using it, and a focus on a shared implementation type system and interface hierarchy between the object and the program that uses it.

SOA - kenmerken

27

- Logical view
- Message orientation
- Description orientation
- Granularity
- Network orientation
- Platform neutral

Industrieel Ingenieur Informatica, Universiteit Gent

A Service Oriented Architecture (SOA) is a form of distributed systems architecture that is typically characterized by the following properties:

Logical view: The service is an abstracted, logical view of actual programs, databases, business processes, etc., defined in terms of what it does, typically carrying out a business-level operation.

Service is abstract gedefinieerd

Los van eigenlijke implementatie

Message orientation: The service is formally defined in terms of the messages exchanged between provider agents and requester agents, and not the properties of the agents themselves. The internal structure of an agent, including features such as its implementation language, process structure and even database structure, are deliberately abstracted away in the SOA: using the SOA discipline one does not and should not need to know how an agent implementing a service is constructed. A key benefit of this concerns so-called legacy systems. By avoiding any knowledge of the internal structure of an agent, one can incorporate any software component or application that can be "wrapped" in message handling code that allows it to adhere to the formal service definition.

Definitie service aan de hand van berichten

Welke berichten verwacht service

Welke antwoordbericht stuurt service

Description orientation: A service is described by machine-processable meta data. The description supports the public nature of the SOA: only those details that are exposed to the public and important for the use of the service should be included in the description. The semantics of a service should be documented, either directly or indirectly, by its description.

Beschrijving service

Leesbaar voor computer

Enkel wat nodig is om de service te gebruiken

Granularity: Services tend to use a small number of operations with relatively large and complex messages.

Beperkt aantal opdrachten

Grote en complex berichten

Network orientation: Services tend to be oriented toward use over a network, though this is not an absolute requirement.

Service aangesproken over netwerk

Niet noodzakelijk

Platform neutral: Messages are sent in a platform-neutral, standardized format delivered through the interfaces. XML is the most obvious format that meets this constraint.

Berichten in gestandariseerd formaat

Meestal XML

Onafhankelijk van een type besturingssysteem, programmeertaal, …

Uitdagingen en problemen SOA

28

- Traagheid en (on)betrouwbaarheid gebruikte transport
- Geen gedeeld geheugen voor requester en provider
- Wat indien een deel van de opdracht mislukt?
- Gelijktijdige toegang tot bronnen
- Wat als één partner incompatibel wordt

Industrieel Ingenieur Informatica, Universiteit Gent

Distributed object systems have a number of architectural challenges.

Problems introduced by latency and unreliability of the underlying transport.

The lack of shared memory between the caller and object.

The numerous problems introduced by partial failure scenarios.

The challenges of concurrent access to remote resources.

The fragility of distributed systems if incompatible updates are introduced to any participant.

COM/CORBA/… versus webservices

29

- Webservices
  - Platformonafhankelijk
  - Over internet
  - Snelheid minder belangrijk
  - Bestaande applicatie openstellen via internet
- COM/CORBA/…
  - Performanter

Industrieel Ingenieur Informatica, Universiteit Gent

These challenges apply irrespective of whether the distributed object system is implemented using COM/CORBA or Web services technologies. Web services are no less appropriate than the alternatives if the fundamental criteria for successful distributed object architectures are met. If these criteria are met, Web services technologies may be appropriate if the benefits they offer in terms of platform/vendor neutrality offset the performance and implementation immaturity issues they may introduce.

Conversely, using Web services technologies to implement a distributed system doesn't magically turn a distributed object architecture into an SOA. Nor are Web services technologies necessarily the best choice for implementing SOAs -- if the necessary infrastructure and expertise are in place to use COM or CORBA as the implementation technology and there is no requirement for platform neutrality, using SOAP/WSDL may not add enough benefits to justify their costs in performance, etc.

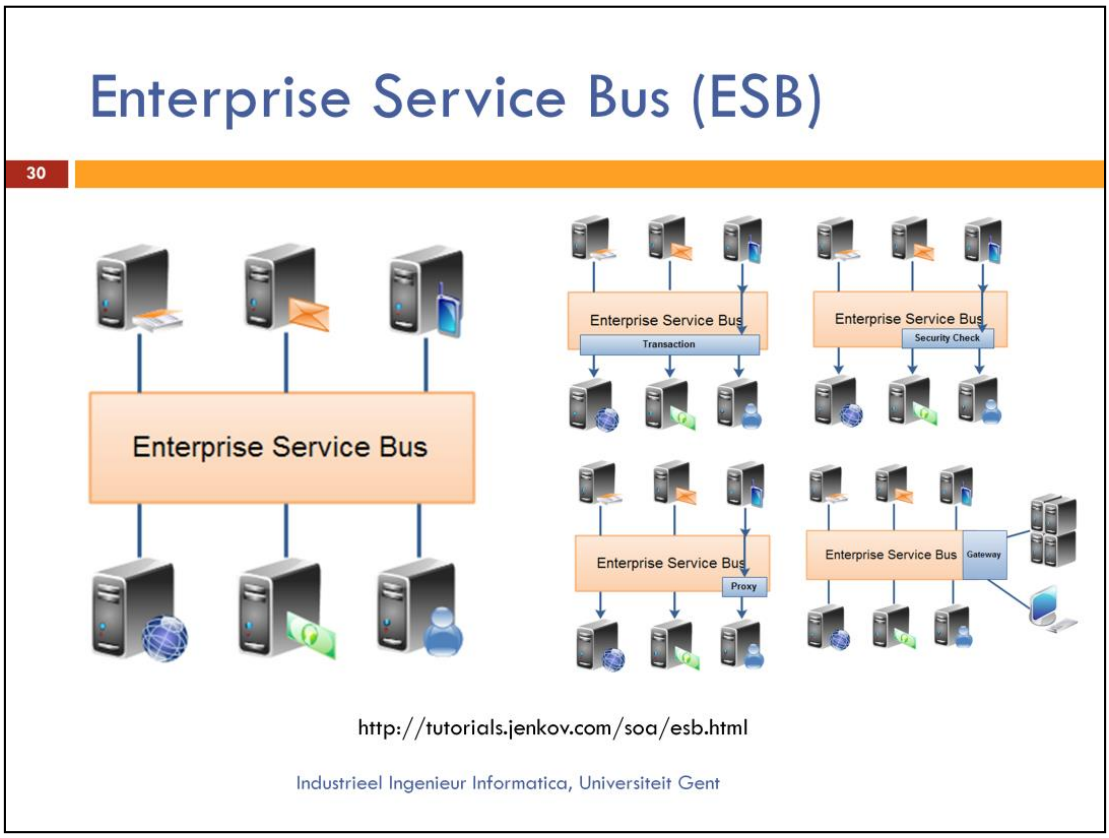
In general SOA and Web services are most appropriate for applications:

That must operate over the Internet where reliability and speed cannot be guaranteed;

Where there is no ability to manage deployment so that all requesters and providers are upgraded at once;

Where components of the distributed system run on different platforms and vendor products;

Where an existing application needs to be exposed for use over a network, and can be wrapped as a Web service.



An "Enterprise Service Bus" (ESB) is a system to which all services are connected. Through the enterprise service bus all connected services can also be accessed.

The term "bus" is an analogy to the internal bus of a computer onto which the CPU, RAM and other chips are connected. An enterprise service bus is typically implemented as a server or a set of servers, and is thus more than just a "network".

Clients and services connected to an enterprise service bus do not communicate directly. They communicate via the ESB. This is done by having the ESB essentially expose the same service interface to potential clients, that the connected services expose to the ESB. This indirection via the ESB has some advantages and disadvantages.

ESB as Single Point of Access

One advantage of connecting clients and services via an enterprise service bus is that clients need only look for services in a single location: The enterprise service bus. If a service is moved from one server to another, you only need to reconfigure the ESB. The clients still just access the service via

the ESB.

ESB as Transaction Manager

Another advantage is that the ESB can coordinate distributed transactions which multiple services participate in. When multiple distributed services need to participate in a transaction some entity typically has to coordinate the transaction. Rather than forcing the client to do this, the enterprise service bus can do so. The client may still have to demarcate the beginning and end of the transaction, even if the work of coordinating the participants is done by the ESB.

ESB as Security Manager

Security aspects like authentication and authorization can be centralized in the enterprise service bus. Even if a service in an application does not have authentication and authorization, the enterprise service bus can require this in the service interface it exposes to potential clients.

ESB as Service Proxy

An ESB may function as a gateway or proxy for applivations that do not expose a standardized service interface to the world. For instance, lets say an application exposes a Java RMI service. The rest of your network is running on .NET which cannot directly call the RMI service.

To solve this problem you can implement a service proxy in Java which can call the RMI service. The service proxy then exposes a web service interface (SOAP + WSDL) via the ESB to the .NET applications.

Such a proxy service does not have to be a built-in capability of an ESB. It can also just be deployed as a separate service, made available via the ESB.

ESB as Gateway to the World

If some clients need to connect to services running in the outside world, the ESB can potentially function as a gateway to the world outside. Again, security aspects etc. can be added ontop of the external service. Furthermore, if the external service is capable of participating in distributed transactions, the ESB can coordinate this too.

ESB Disadvantages

Connecting your services via an ESB also has a few disadvantages. First of all the ESB may become a single point of failure. If the ESB is down, no communication between clients and services can take place. Second, the extra level of indirection may result in decreased performance of client-service communication.

Gedistribueerde applicaties

31

- EDI
- RPC
- RMI, DCOM, CORBA
- Webservices
  - Definitie
  - Architectuur
  - Technologieën
    - SOAP

Twee types webservices

32

- RESTfull webservices
  - Roy Fielding
  - JAX-RS (Java API for RESTful Web Services)
  - Web API (.NET)
- SOAP webservices

Industrieel Ingenieur Informatica, Universiteit Gent

The World Wide Web operates as a networked information system that imposes several constraints: Agents identify objects in the system, called resources, with Uniform Resource Identifiers (URIs). Agents represent, describe, and communicate resource state via representations of the resource in a variety of widely-understood data formats (e.g. XML, HTML, CSS, JPEG, PNG). Agents exchange representations via protocols that use URIs to identify and directly or indirectly address the agents and resources.

An even more constrained architectural style for reliable Web applications known as Representation State Transfer (REST) has been proposed by Roy Fielding and has inspired both the W3C Technical Architecture Group's architecture document and many who see it as a model for how to build Web services. The REST Web is the subset of the WWW (based on HTTP) in which agents provide uniform interface semantics -- essentially create, retrieve, update and delete -- rather than arbitrary or application-specific interfaces, and manipulate resources only by the exchange of representations. Furthermore, the REST interactions are "stateless" in the sense that the meaning of a message does not depend on the state of the conversation.

We can identify two major classes of Web services:

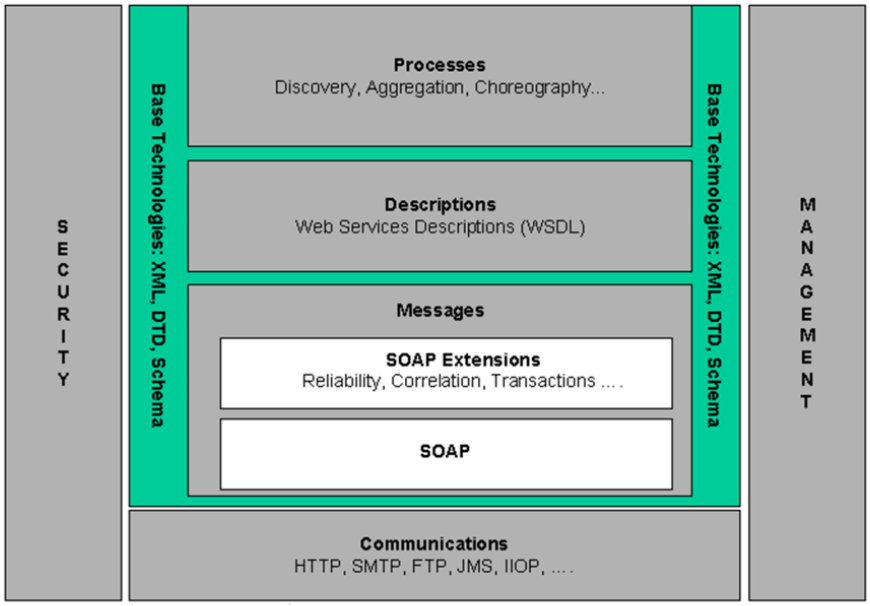
REST-compliant Web services, in which the primary purpose of the service is to manipulate XML representations of Web resources using a uniform set of "stateless" operations; and

arbitrary Web services, in which the service may expose an arbitrary set of operations.

Both classes of Web services use URIs to identify resources and use Web protocols (such as HTTP and SOAP 1.2) and XML data formats for messaging. (It should be noted that SOAP 1.2 can be used in a manner consistent with REST. However, SOAP 1.2 can also be used in a manner that is not consistent with REST.)

Technologieën

33



http://www.w3.org/TR/ws-arch/

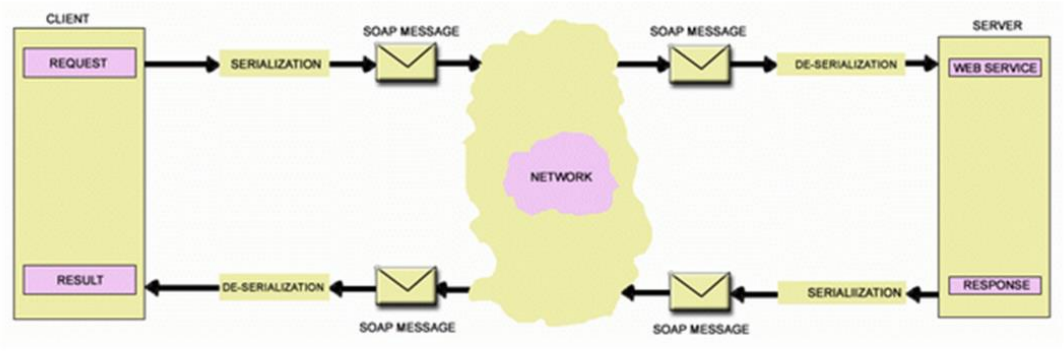
A choreography is a model of the sequence of operations, states, and conditions that control the interactions involved in the participating services. The interaction prescribed by a choreography results in the completion of some useful function. Examples include the placement of an order, information about its delivery and eventual payment, or putting the system into a well-defined error state.

A choreography can be distinguished from an orchestration. An orchestration defines the sequence and conditions in which *one* Web service invokes other Web services in order to realize some useful function.

A choreography may be described using a choreography description language. A choreography description language permits the description of how Web services can be composed, how service roles and associations in Web services can be established, and how the state, if any, of composed services is to be managed.

SOAP webservices

34



http://www.codeproject.com/Articles/6399/Securing-web-services-using-SOAP-extensions

Industrieel Ingenieur Informatica, Universiteit Gent

Framework

- Protocol gebaseerd op XML

Uitwisseling van informatie in een gedecentraliseerde, gedistribueerde omgeving

- Versturen van XML-berichten

Transport via het web (HTTP, …)

Betekenis

- Simple Object Acces Protocol
- Service Oriented Architecture Protocol

SOAP specificatie omvat

XML-envelop

- XML-tags die een SOAP-bericht beschrijven en die een framework geven voor de inhoud van het bericht
- Met XML-inhoud
  - Conventies voor het uitvoeren van methodes op een andere server en het voorstellen van het resultaat

Een aantal regels voor servers die een SOAP-boodschap ontvangen

Maakt gebruik van bestaande transportprotocols (bv. HTTP)

Oorspronkelijk Microsoft, ondertussen standaard van W3C

SOAP 1.2 provides a standard, extensible, composable framework for packaging and exchanging XML messages. In the context of this architecture, SOAP 1.2 also provides a convenient mechanism for referencing capabilities (typically by use of headers).

[SOAP 1.2 Part 1] defines an XML-based messaging framework: a processing model and an extensibility model. SOAP messages can be carried by a variety of network protocols; such as HTTP, SMTP, FTP, RMI/IIOP, or a proprietary messaging protocol.

[SOAP 1.2 Part 2] defines three optional components: a set of encoding rules for expressing instances of application-defined data types, a convention for representing remote procedure calls (RPC) and responses, and a set of rules for using SOAP with HTTP/1.1.

While SOAP Version 1.2 [SOAP 1.2 Part 1] doesn't define "SOAP" as an acronym anymore, there are two expansions of the term that reflect these different ways in which the technology can be interpreted:

Service Oriented Architecture Protocol: In the general case, a SOAP message represents the information needed to invoke a service or reflect the results of a service invocation, and contains the information specified in the service interface definition.

Simple Object Access Protocol: When using the optional SOAP RPC Representation, a SOAP message represents a method invocation on a remote object, and the serialization of in the argument list of that method that must be moved from the local environment to the remote environment.

A SOAP message is fundamentally a one-way transmission between SOAP nodes, from a SOAP sender to a SOAP receiver, but SOAP messages are expected to be combined by applications to implement more complex interaction patterns ranging from request/response to multiple, back-and-forth "conversational" exchanges.

Voorbeeld SOAP-aanvraag

35

```
POST /ZwiftBooks HTTP/1.1
Host: www.zwiftbooks.com
Content-Type:text/xml
Content-Length: 134
SOAP Action: “Some-URI”

<Envelope xmlns=“http://www.w3.org/2003/05/soap-envelope”
   encodingStyle=“http://www.w3.org/2003/05/soap-encoding”>
    <Body>
       <zwiftbooks:GetGuaranteedDeliveryTime
         xmlns:zwiftbooks=“www.zwiftbooks.com”>
           <zwiftbooks:isbn>0-13-18188-222-2</zwiftbooks:isbn>
           <zwiftbooks:zipcode>75240</zwiftbooks:zipcode>
       </zwifbooks:GetGuaranteedDeliveryTime>
    </Body>
</Envelope>
```

Industrieel Ingenieur Informatica, Universiteit Gent

De eerste lijn en de headerlijnen blijven bestaan

Het corpus van een HTTP-aanvraag en HTTP-antwoord bevat XML ipv HTML, …

Analoog via FTP of SMTP

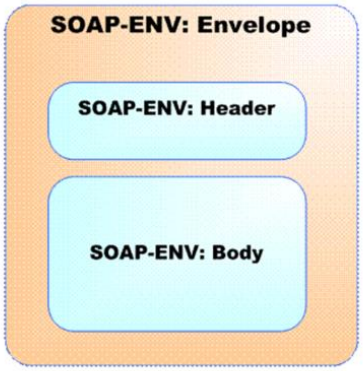
Voorbeeld SOAP-antwoord

36

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 122

<Envelope xmlns="http://www.w3.org/2003/05/soap-
  envelope">
  <Body>
    <zwiftbooks:GetBestDeliveryTimeResponse
     xmlns:zwiftbooks="www.zwiftbooks.com">
      <zwiftbooks:Time>2 hours</zwiftbooks:Time>
    </zwiftbooks:GetBestDeliveryTimeResponse>
  </Body>
</Envelope>
```

Structuur SOAP Bericht

37



https://mytechprep.wordpress.com/tag/soap-actor/

Industrieel Ingenieur Informatica, Universiteit Gent

SOAP-envelop:
- Envelope-element
  - Attributen: encodingStyle: hoe geserialiseerd wordt
- Rootelement van het XML-document dat een SOAP-bericht voorstelt
- Verplicht

SOAP-hoofding:
- Header-element
- Extra richtlijnen en informatie voor SOAP-server

SOAP-corpus:
- Body-element
- Bevat de uit te wisselen XML
  - Onafhankelijk van SOAP-structuur
  - Domein-specifieke structuur
- Inhoud
  - Data/Resultaat
  - Aanroep van een methode
- Verplicht

Voorbeeld: SOAP-hoofding

38

```
<header>
   <n:alertcontrol
     xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
   </n:alertcontrol>
</header>
```

Industrieel Ingenieur Informatica, Universiteit Gent

SOAP-hoofding:

- Header-element
- Extra richtlijnen en informatie voor SOAP-server
  - Transacties
  - Beveiliging
  - Filters
  - …
- Optioneel

SOAP-berichtenpad

39

https://myshadesofgray.wordpress.com/category/programming/java/

Industrieel Ingenieur Informatica, Universiteit Gent

Bestaat uit SOAP-knopen:
- SOAP-zender
- Nul of meerdere tussenliggende SOAP-servers
- SOAP-ontvanger

Tussenliggende SOAP-servers
- Stuurt SOAP-bericht door
- Behandelt de voor hem bedoelde SOAP-hoofdingen

Elke SOAP-knoop heeft één of meerdere rollen

Mogelijke rollen
- Gespecifieerd door SOAP:
  - none (niet bedoeld voor een specifieke knoop)
  - next (ontvanger en alle tussenliggende)
  - ultimateReceiver (ontvanger)
- Gespecifieerd door de applicatie
  - Proxy
  - Beveiliging
  - Caching
  - …

Voorbeeld: SOAP-hoofding

40

```
<Header role="http://www.w3.org/2003/05/soap-
  envelope/role/next">
  <n:alertcontrol
    xmlns:n="http://example.org/alertcontrol"
    mustUnderstand="true">
    <n:priority>1</n:priority>
    <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
</Header>
```

Industrieel Ingenieur Informatica, Universiteit Gent

Een SOAP-hoofding kan aangeven voor welke SOAP-knoop de hoofding bedoeld is door gebruik te maken van rollen

- Attribuut role
- Waarde attribuut
  - Rol gespecifieerd door SOAP
  - URI SOAP-knoop

Regels voor SOAP-server

Identificatie van de stukken van het SOAP-bericht die voor de huidige server bedoeld zijn

Nagaan of deze server alle delen van de headers die bedoeld zijn voor deze server en die het attribuut mustUnderstand hebben, ondersteunt. Indien niet fout genereren

Uitvoeren van de stukken van de headers bedoeld voor deze server

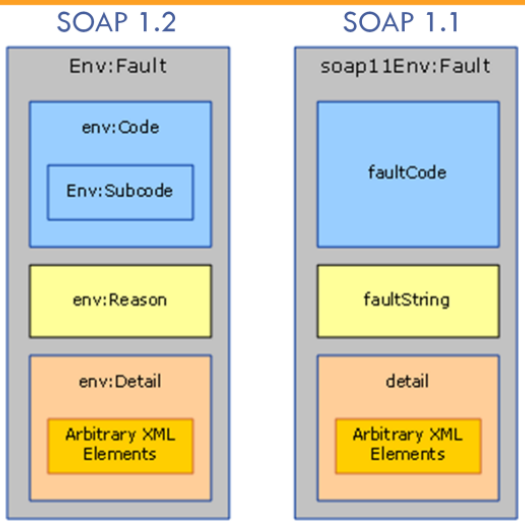
Indien niet de SOAP-ontvanger: alle headers bedoeld voor deze server verwijderen en bericht doorsturen

SOAP headers have been designed in anticipation of various uses for SOAP, many of which will involve the participation of other SOAP processing nodes - called SOAP intermediaries - along a message's path from an initial

SOAP sender to an ultimate SOAP receiver. This allows SOAP intermediaries to provide value-added services. Headers may be inspected, inserted, deleted or forwarded by SOAP nodes encountered along a SOAP message path. (It should be kept in mind, though, that the SOAP specifications do not deal with what the contents of header elements are, or how SOAP messages are routed between nodes, or the manner by which the route is determined and so forth. These are a part of the overall application, and could be the subject of other specifications.)

SOAP-fouten

41



https://msdn.microsoft.com/en-us/library/ms789039%28v=vs.110%29.aspx

Wanneer

- Niet begrijpen bericht
- Fout bij behandelen van bericht

fault-element

- Kind van body-element
- Kinderen (SOAP 1.1)
  - faultcode: foutcode
  - faultstring: leesbare verklaring
  - detail: informatie over het probleem

SOAP 1.2 provides a model for handling situations when faults arise in the processing of a message. SOAP distinguishes between the conditions that result in a fault, and the ability to signal that fault to the originator of the faulty message or another node. The ability to signal the fault depends on the message transfer mechanism used, and one aspect of the binding specification of SOAP onto an underlying protocol is to specify how faults are signalled, if at all. The remainder of this section assumes that a transfer mechanism is available for signalling faults generated while processing received messages, and concentrates on the structure of the SOAP fault message.

The SOAP env:Body element has another distinguished role in that it is the place where such fault information is placed. The SOAP fault model requires that all SOAP-specific and application-specific faults be reported using a single distinguished element, env:Fault, carried within the env:Body element. The env:Fault element contains two mandatory sub-elements, env:Code and env:Reason, and (optionally) application-specific information in the env:Detail sub-element. Another optional sub-element, env:Node, identifies via a URI the SOAP node which generated the fault, its absence implying that it was the ultimate recipient of the message which did so. There is yet another optional sub-element, env:Role, which identifies the role being played by the node which generated the fault.

The env:Code sub-element of env:Fault is itself made up of a mandatory env:Value sub-element, whose content is specified in the SOAP specification (see SOAP Part 1 section 5.4.6) as well as an optional env:Subcode sub-element.

Voorbeeld SOAP-fouten

42

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env= “” xmlns:rpc='http://www.w3.org/2003/05/soap-rpc'>
  <env:Body> <env:Fault>
     <env:Code>
       <env:Value>env:Sender</env:Value>
       <env:Subcode><env:Value>rpc:BadArguments</env:Value></env:Subcode></env:Co
     <env:Reason>
      <env:Text xml:lang="en-US">Processing error</env:Text></env:Reason>
     <env:Detail>
      <e:myFaultDetails xmlns:e="http://travelcompany.example.org/faults">
        <e:message>Name does not match card number</e:message><e:errorcode>999</e
      </e:myFaultDetails>
     </env:Detail>
   </env:Fault> </env:Body>
</env:Envelope>
```

Gedistribueerde applicaties

43

- EDI
- RPC
- RMI, DCOM, CORBA
- Webservices
  - Definitie
  - Architectuur
  - Technologieën
    - SOAP
    - WSDL

WSDL

44

- Web Service Description Language
- XML-standaard W3C
- Twee versies
  - WSDL 1.1 (http://www.w3.org/TR/2001/NOTE-wsdl-20010315)
  - WSDL 2.0 (http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/)

Industrieel Ingenieur Informatica, Universiteit Gent

Beschrijven webservice

Interface

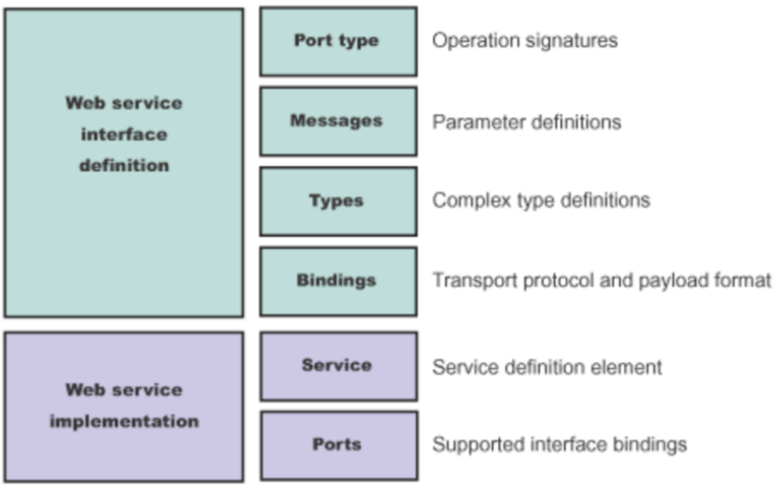
Implementatiedetails

Bepaalt hoe de service requester de aangeboden dienst (service provider) moet aanspreken

Web Services Description Language (WSDL) provides a model and an XML format for describing Web services. WSDL enables one to separate the description of the abstract functionality offered by a service from concrete details of a service description such as "how" and "where" that functionality is offered.

WSDL 1.1

45



https://www-01.ibm.com/support/knowledgecenter/SSAW57_8.0.0/com.ibm.websphere.nd.doc/info/ae/ae/cwbs_wsdl.html

Specificatie interface webservice

- Leesbaar door computers

Definieert

- Formaat berichten
- Datatypes
- Transportprotocollen
- Serialisatieformaten voor transport
- Netwerklocaties
- Patroon om berichten uit te wisselen

Types: Type-declaraties van argumenten, methodes, teruggeefwaardes, … (XML Schema)

Message: Beschrijving bericht (aanvraag/antwoord)

PortType: Beschrijving mogelijke opdracht (naam, input, output)

Binding: Verbinden portType met een locatie en een protocol (SOAP, HTTP GET, …)

Service: Uit welke portTypes bestaat de webservice

As communications protocols and message formats are standardized in the

web community, it becomes increasingly possible and important to be able to describe the communications in some structured way. WSDL addresses this need by defining an XML grammar for describing network services as collections of communication endpoints capable of exchanging messages. WSDL service definitions provide documentation for distributed systems and serve as a recipe for automating the details involved in applications communication.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

Types– a container for data type definitions using some type system (such as XSD).

Message– an abstract, typed definition of the data being communicated.

Operation– an abstract description of an action supported by the service.

Port Type–an abstract set of operations supported by one or more endpoints.

Binding– a concrete protocol and data format specification for a particular port type.

Port– a single endpoint defined as a combination of a binding and a network address.

Service– a collection of related endpoints.

It is important to observe that WSDL does not introduce a new type definition language. WSDL recognizes the need for rich type systems for describing message formats, and supports the XML Schemas specification (XSD) as its canonical type system. However, since it is unreasonable to expect a single type system grammar to be used to describe all message formats present and future, WSDL allows using other type definition languages via extensibility.

In addition, WSDL defines a common binding mechanism. This is used to attach a specific protocol or data format or structure to an abstract message, operation, or endpoint. It allows the reuse of abstract definitions.

In addition to the core service definition framework, this specification introduces specific binding extensions for the following protocols and message formats:

SOAP

HTTP GET / POST

MIME

Voorbeeld WSDL 1.1


```
<?xml version="1.0"?>
<definitions name="StockQuote“
   targetNamespace=http://example.com/stockquote.wsdl
   xmlns:tns=http://example.com/stockquote.wsdl
   xmlns:xsd1=http://example.com/stockquote.xsd
   xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
   xmlns="http://schemas.xmlsoap.org/wsdl/">
    <types>
       <schema targetNamespace=http://example.com/stockquote.xsd
        xmlns="http://www.w3.org/2000/10/XMLSchema">
          <element name="TradePriceRequest">
             <complexType>…</complexType>
          </element>
          <element name="TradePrice">
             <complexType> …</complexType>
          </element>
       </schema>
    </types>
```

Voorbeeld WSDL 1.1

47

```
<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
       <input message="tns:GetLastTradePriceInput"/>
       <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
```

Industrieel Ingenieur Informatica, Universiteit Gent

WSDL 1.1 has four transmission primitives that an endpoint can support:

One-way. The endpoint receives a message.

```
<wsdl:operation name="nmtoken">
  <wsdl:input name="nmtoken"? message="qname"/>
</wsdl:operation>
```

Request-response. The endpoint receives a message, and sends a correlated message.

```
<wsdl:operation name="nmtoken" parameterOrder="nmtokens">
  <wsdl:input name="nmtoken"? message="qname"/>
  <wsdl:output name="nmtoken"? message="qname"/>
  <wsdl:fault name="nmtoken" message="qname"/>*
</wsdl:operation>
```

Solicit-response. The endpoint sends a message, and receives a correlated message.

```
<wsdl:operation name="nmtoken" parameterOrder="nmtokens">
  <wsdl:output name="nmtoken"? message="qname"/>
  <wsdl:input name="nmtoken"? message="qname"/>
  <wsdl:fault name="nmtoken" message="qname"/>*
```

Notification. The endpoint sends a message.

```
<wsdl:operation name="nmtoken">
  <wsdl:output name="nmtoken"? message="qname"/>
</wsdl:operation>
```

Although the base WSDL 1.1 structure supports bindings for these four transmission primitives, WSDL only defines bindings for the One-way and Request-response primitives. It is expected that specifications that define the protocols for Solicit-response or Notification would also include WSDL binding extensions that allow use of these primitives.

Voorbeeld WSDL 1.1

48

```
<binding name="StockQuoteSoapBinding"
   type="tns:StockQuotePortType">
    <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetLastTradePrice">
       <soap:operation
   soapAction="http://example.com/GetLastTradePrice"/>
       <input>
           <soap:body use="literal"/>
       </input>
       <output>
           <soap:body use="literal"/>
       </output>
    </operation>
</binding>
```

Industrieel Ingenieur Informatica, Universiteit Gent

Style: document, rpc

The style attribute indicates whether the operation is RPC-oriented (messages containing parameters and return values) or document-oriented (message containing document(s)). This information may be used to select an appropriate programming model. The value of this attribute also affects the way in which the Body of the SOAP message is constructed. If the attribute is not specified, it defaults to the value specified in the soap:binding element. If the soap:binding element does not specify a style, it is assumed to be "document".

Alternatieve bindings: HTTP GET & POST Binding

The soapAction attribute specifies the value of the SOAPAction header for this operation. This URI value should be used directly as the value for the SOAPAction header; no attempt should be made to make a relative URI value absolute when making the request. For the HTTP protocol binding of SOAP, this is value required (it has no default value). For other SOAP protocol bindings, it MUST NOT be specified, and the soap:operation element MAY be omitted.

The soap:body element specifies how the message parts appear inside the SOAP Body element.

The parts of a message may either be abstract type definitions, or concrete schema definitions. If abstract definitions, the types are serialized according to some set of rules defined by an encoding style. Each encoding style is identified using a list of URIs, as in the SOAP specification. Since some encoding styles such as the SOAP Encoding (http://schemas.xmlsoap.org/soap/encoding/) allow variation in the message format for a given set of abstract types, it is up to the reader of the message to understand all the format variations: "reader makes right". To avoid having to support all variations, a message may be defined concretely and then indicate it's original encoding style (if any) as a hint. In this case, the writer of the message must conform exactly to the specified schema: "writer makes right".

The required **use** attribute indicates whether the message parts are encoded using some encoding rules, or whether the parts define the concrete schema of the message.

If use is **encoded**, then each message part references an abstract type using the **type** attribute. These abstract types are used to produce a concrete message by applying an encoding specified by the **encodingStyle** attribute. The part **names**, **types** and value of the **namespace** attribute are all inputs to the encoding, although the namespace attribute only applies to content not explicitly defined by the abstract types. If the referenced encoding style allows variations in it's format (such as the SOAP encoding does), then all variations MUST be supported ("reader makes right").

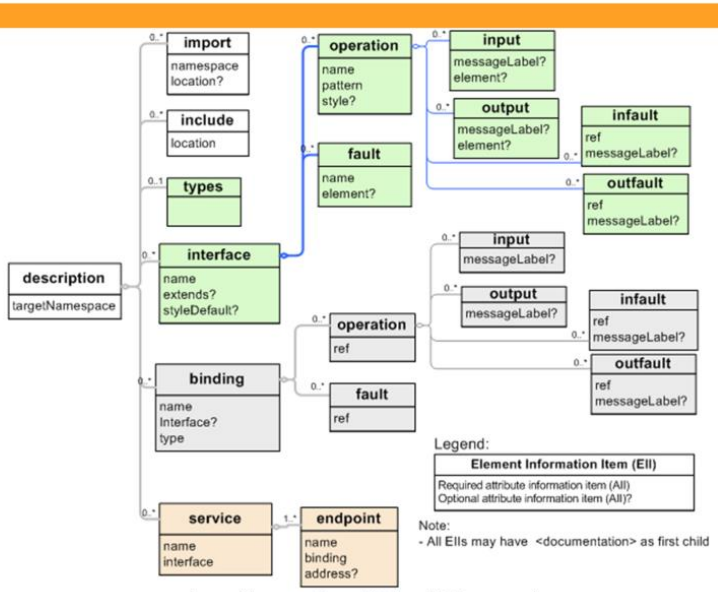
If use is **literal**, then each part references a concrete schema definition using either the **element** or **type** attribute. In the first case, the element referenced by the part will appear directly under the Body element (for document style bindings) or under an accessor element named after the message part (in rpc style). In the second, the type referenced by the part becomes the schema type of the enclosing element (Body for document style or part accessor element for rpc style).

Voorbeeld WSDL 1.1


```
    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
           <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>

</definitions>
```

WSDL 2.0 -structuur

50



http://www.w3.org/TR/wsdl20-primer/

Industrieel Ingenieur Informatica, Universiteit Gent

types: beschrijving berichten in XML Schema

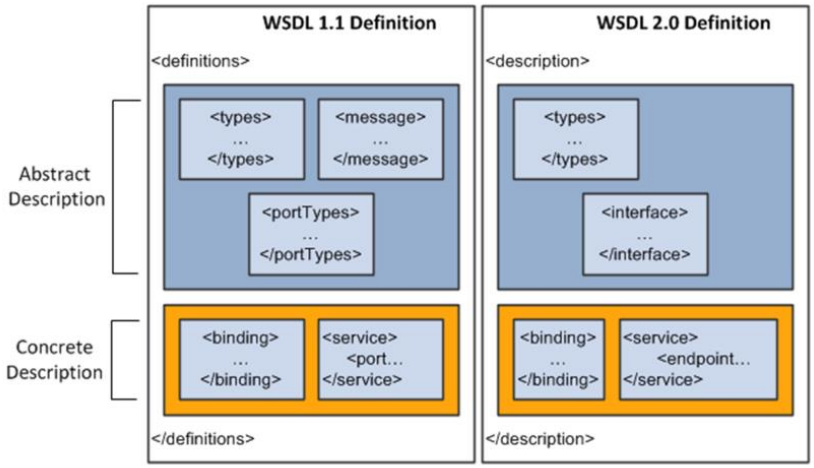
interface: abstracte funcionaliteit

binding: hoe toegang tot diensten

service: waar toegang tot diensten

WSDL 1.0 versus 2.0

51



http://docs.oracle.com/cd/E41633_01/pt853pbh1/eng/pt/tibr/concept_UnderstandingProvidingWSDLDocuments-076201.htm

Voorbeeld WSDL 2.0

52

```
<?xml version="1.0" encoding="utf-8" ?>
<description xmlns="http://www.w3.org/ns/wsdl"
   targetNamespace=http://greath.example.com/2004/wsdl/resSvc
   xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc" …>
   …
</description>
```

Voorbeeld WSDL 2.0

53

```
<description  …> …
 <types>
  <xs:schema …>
    <xs:element name="checkAvailability" type="tCheckAvailability"/>

      <xs:complexType name="tCheckAvailability">
        <xs:sequence>
         <xs:element name="checkInDate" type="xs:date"/>
         <xs:element name="checkOutDate" type="xs:date"/>
         <xs:element name="roomType" type="xs:string"/>
       </xs:sequence> </xs:complexType>
    <xs:element name="checkAvailabilityResponse" type="xs:double"/>
    <xs:element name="invalidDataError" type="xs:string"/>
 </xs:schema>
 </types> …
</description>
```

Voorbeeld WSDL 2.0

54

```
<description …> … <types> … </types>
  <interface name="reservationInterface" >
    <fault name="invalidDataFault" element="ghns:invalidDataError"/>
    <operation name="opCheckAvailability"
      pattern="http://www.w3.org/ns/wsdl/in-out"
      style="http://www.w3.org/ns/wsdl/style/iri" wsdlx:safe = "true">
        <input messageLabel="In" element="ghns:checkAvailability" />
        <output messageLabel="Out"
           element="ghns:checkAvailabilityResponse" />
        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
    </operation>
  </interface> … </description>
```

Industrieel Ingenieur Informatica, Universiteit Gent

WSDL 2.0 enables one to separate the description of a Web service's abstract functionality from the concrete details of how and where that functionality is offered. This separation facilitates different levels of reusability and distribution of work in the lifecycle of a Web service and the WSDL 2.0 document that describes it.

A WSDL 2.0 interface defines the abstract interface of a Web service as a set of abstract operations, each operation representing a simple interaction between the client and the service. Each operation specifies the types of messages that the service can send or receive as part of that operation. Each operation also specifies a message exchange pattern that indicates the sequence in which the associated messages are to be transmitted between the parties. For example, the in-out pattern indicates that if the client sends a message in to the service, the service will either send a reply message back out to the client (in the normal case) or it will send a fault message back to the client (in the case of an error).

style="http://www.w3.org/ns/wsdl/style/iri“ This line indicates that the XML schema defining the input message of this operation follows a set of rules as specified in IRI Style that ensures the message can be serialized as an IRI. When using this style, the value of the {message content model} property of the Interface Message Reference component corresponding to

the initial message of the message exchange pattern MUST be "#element".†

Use of this value indicates that XML Schema [XML Schema Structures] was used to define the schema of the {element declaration} property of the Interface Message Reference component of the Interface Operation component corresponding to the initial message of the message exchange pattern.

wsdlx:safe="true" > This line indicates that this operation will not obligate the client in any way, i.e., the client can safely invoke this operation without fear that it may be incurring an obligation (such as agreeing to buy something).

<input messageLabel="In" The input element specifies an input message. Even though we have already specified which message exchange pattern the operation will use, a message exchange pattern represents a template for a message sequence, and in theory could consist of multiple input and/or output messages. Thus we must also indicate which potential input message in the pattern this particular input message represents. This is the purpose of the messageLabel attribute. Since the in-out pattern that we've chosen to use only has one input message, it is trivial in this case: we simply fill in the message label "In" that was defined in WSDL 2.0. However, if a new pattern is defined that involve multiple input messages, then the different input messages in the pattern could then be distinguished by using different labels.

<output messageLabel="Out" . . . This is similar to defining an input message.

Voorbeeld WSDL 2.0

55

```
<description … > … <types> … </types>
<interface name = "reservationInterface" > … </interface>
<binding name="reservationSOAPBinding"
   interface="tns:reservationInterface"
   type="http://www.w3.org/ns/wsdl/soap"
   wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
   <operation ref="tns:opCheckAvailability"
      wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"/>
   <fault ref="tns:invalidDataFault" wsoap:code="soap:Sender"/>
</binding>
. . .
</description>
```

Industrieel Ingenieur Informatica, Universiteit Gent

Although we have specified what abstract messages can be exchanged with the GreatH Web service, we have not yet specified how those messages can be exchanged. This is the purpose of a binding. A binding specifies concrete message format and transmission protocol details for an interface, and must supply such details for every operation and fault in the interface.

In the general case, binding details for each operation and fault are specified using operation and fault elements inside a binding element.

interface="tns:reservationInterface“ This is the name of the interface whose message format and transmission protocols we are specifying

type="http://www.w3.org/ns/wsdl/soap" This specifies what kind of concrete message format to use, in this case SOAP 1.2

wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/“ This attribute is specific to WSDL 2.0's SOAP binding extension (thus it uses the wsoap: prefix). It specifies the underlying transmission protocol that should be used, in this case HTTP.

<operation ref="tns:opCheckAvailability“ This is not defining a new operation; rather, it is referencing the previously defined opCheckAvailability operation in order to specify binding details for it.

wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response"> This attribute is also specific to WSDL 2.0's SOAP binding extension. It specifies the SOAP message exchange pattern (MEP) that will be used to implement the abstract WSDL 2.0 message exchange pattern (in-out) that was specified when the opCheckAvailability operation was defined.

One-way.

Request-response

Duplex

When HTTP is used as the underlying transport protocol (as in this example) the wsoap:mep attribute also controls whether GET or POST will be used as the underlying HTTP method. In this case, the use of wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-response" causes GET to be used by default.

<fault ref="tns:invalidDataFault“ As with a binding operation, this is not declaring a new fault; rather, it is referencing a fault (invalidDataFault) that was previously defined in the opCheckAvailability interface, in order to specify binding details for it.

wsoap:code="soap:Sender"/> This attribute is also specific to WSDL 2.0's SOAP binding extension. This specifies the SOAP 1.2 fault code that will cause this fault message to be sent. If desired, a list of subcodes can also be specified using the optional wsoap:subcodes attribute.

Voorbeeld WSDL 2.0

56

```
<?xml version="1.0" encoding="utf-8" ?>
<description …> … <types> … </types>
 <interface name = "reservationInterface" > … </interface>
 <binding name="reservationSOAPBinding"
   interface="tns:reservationInterface" …> ... </binding>
 <service name="reservationService"
   interface="tns:reservationInterface">
   <endpoint name="reservationEndpoint"
      binding="tns:reservationSOAPBinding"
      address ="http://greath.example.com/2004/reservation"/>
 </service>
</description>
```

Industrieel Ingenieur Informatica, Universiteit Gent

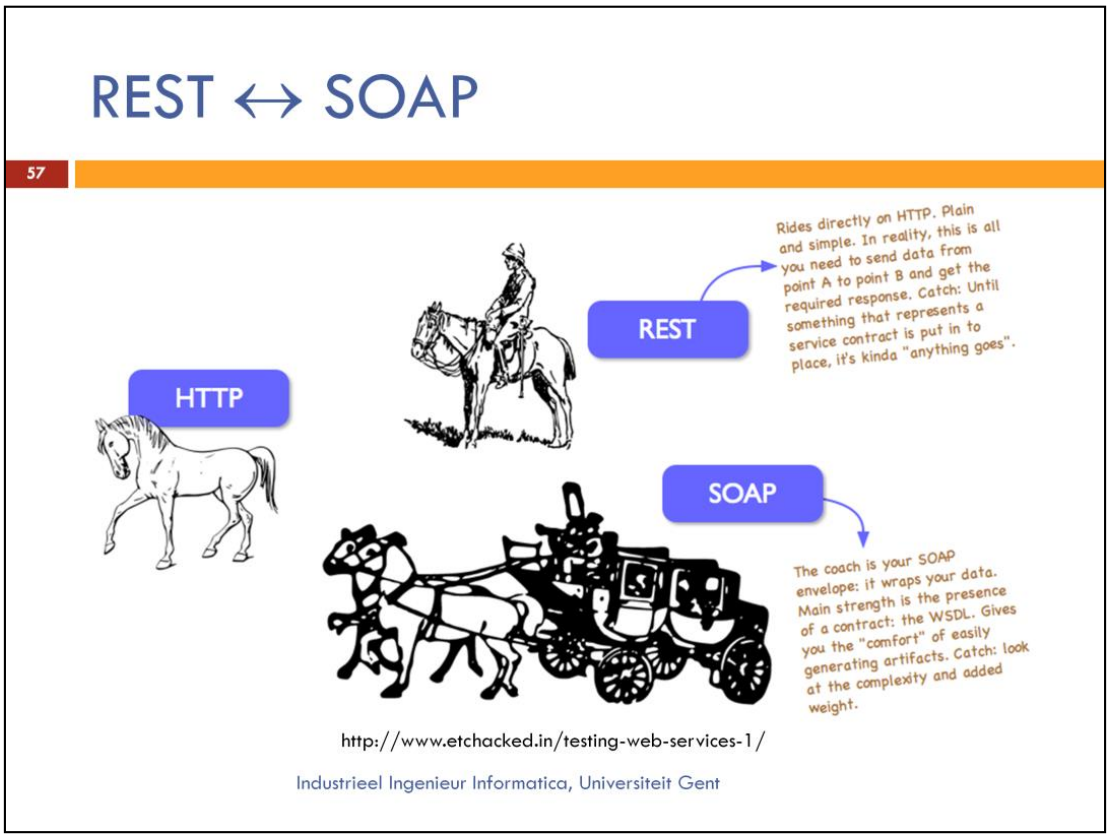
<service name="reservationService“ This defines a name for this service, which must be unique among service names in the WSDL 2.0 target namespace. The name attribute is required.

interface="tns:reservationInterface"> This specifies the name of the previously defined interface that these service endpoints will support.

<endpoint name="reservationEndpoint“ This defines an endpoint for the service, and a name for this endpoint, which must be unique within this service.

binding="tns:reservationSOAPBinding“ This specifies the name of the previously defined binding to be used by this endpoint.

address ="http://greath.example.com/2004/reservation"/> This specifies the physical address at which this service can be accessed using the binding specified by the binding attribute.



REST

- Volledig statusloos
- Caching verhoogt performantie
- Service en gebruiker service kennen de context (structuur berichten, …)
  - Geen formele beschrijving zoals WSDL
- Beperkte bandbreedte (bv. Mobiele toestellen)
- Invoegen in bestaande website met bv. Ajax

SOAP

- Formeel contract tussen gebruiker en service (WSDL)
- Complexere functionaliteit
  - Meer dan CRUD
  - Beveiliging, transacties, …
- Asynchrone afhandeling