1 Cooperating Programs – Why?

What's Wrong – Reliability

- Business loses millions of dollars every minute the server is down.
- Have you ever tried to run server? How much downtime did you have?
- Critical systems need 99.999 % reliability = 5 minutes/year.
- Examples of failure: "České spořitelně v sobotu několik hodin nefungovalo internetové bankovnictví."
- Amazon cloud 2017: `https://en.wikipedia.org/wiki/Timeline\_of\_Amazon\_Web\_Services\#Amazon\_Web\_Services\_outages`
- Solution: Backup systems
- Problem: double/triple price, same performance

What's Wrong – Scaling

- Hardware doesn't scale well
- RAM scaling:
  - 16 GB CZK 800
  - 32 GB CZK 1.500
  - 64 GB CZK 5.000
  - 128 GB CZK 7-14.000
  - 256 GB CZK 100.000
  - 512 GB CZK 300.000
  - 10 TB? How? Mainframe? Great for very rich customers.
- The same problem is with disks (RAID helps a bit), CPUs...

Solution – Horizontal Scaling

- Let's use backup system to cooperate on processing data!
- Let's have multiple **cheap** computers, where price of 1 TB RAM = 16 × 64 GB, CZK 80.000 (compare to 2 × 512 GB, 600.000)
- Similar approach as RAID (Redundant Array of Inexpensive Disks)
- How to distribute the tasks?

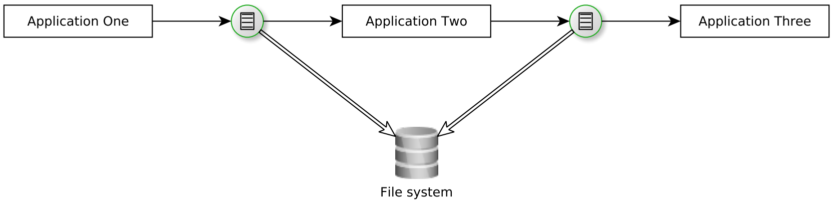


Figure 1: Application pipeline diagram.

Distributed Systems

- Distributed (fault tolerant) systems
  - Able to process requests concurrently
  - Scalable
  - Can handle faults, only decrease performance
- Caveats
  - Less predictable
  - More complex
  - More difficult to secure
  - Effort to manage the system

2 Approaches – Data Sharing

File

- Applications exchange data by writing into a shared file
- Pipeline processing
- Shared filesystems, locking
- Problems: format, schema, scalability, concurrency, notifications

Database

- Applications share database, possibly use different views of the same database
- No integration layer needed, application data always up to date
- Problems: polling – no notifications, schema evolution, still used!

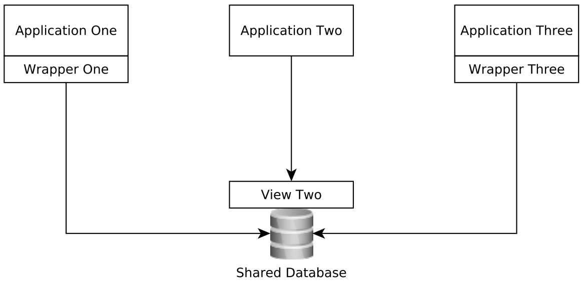


Figure 2: Applications using shared database.

3 Remote Execution – Platform-specific

RPC

- *Remote Procedure Call*, theory in 1970s, first implementation in early 1982, started in Unix/C, today NFS, known target
- Client invokes methods of a *remote interface* on a local *stub*
  - Stub is a RMI-generated *proxy* object representing the remote implementation
- Server implements *remote interface* to export methods which can be called remotely
- Object-oriented equivalent of *remote procedure call* (see later)
- Java – *RMI*, Python – *RPyC*, Ruby – *Distributed Ruby*, Erlang – built into the language, Go, Rust – *Tarpc*

Java RMI

- Java-specific technology for distributed systems
- Java Remote Method Protocol
  - Wire-level protocol (application layer) on top of TCP
  - Binary
- RMI supports primitive types and `Serializable`
- RMI registry
  - Server registers at RMI registry as a provider of remote objects
  - Client uses RMI registry to look up remote objects

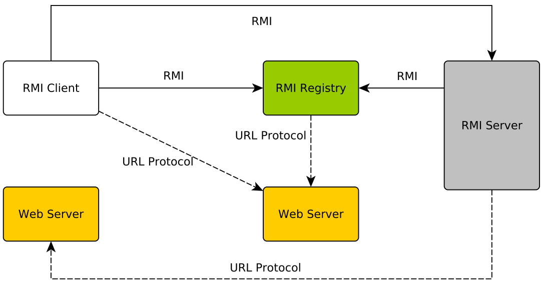


Figure 3: Schema of Java RMI components.

4 Remote Execution – Platform-independent

XML-RPC

- Client-server architecture
- Typically synchronous
- Try it Yourself: `https://gitlab.fel.cvut.cz/ear/xmlrpcserver`

XML-RPC

- Standard for remote procedure call using XML as message format
- Platform independent
- Over HTTP

XML-RPC Example

Request

```
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><int>41</int></value>
    </param>
  </params>
</methodCall>
```

Response

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

CORBA

- *Common Object Request Broker Architecture*
- Introduces Middleware (ORB), more complex setup
- OMG standard for language and platform-independent distributed computing architecture
- Similar to RPC but object-oriented
- Transparent location – client is unaware whether invocation is local or remote
  - Also a caveat – local invocation cannot be optimized and has to go through the whole ORB machinery
- Standards for interface definition, communication protocols, location

CORBA – IDL

Interface Definition Language (IDL)

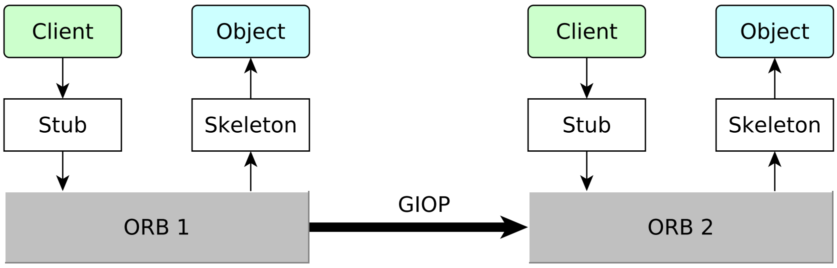
- Standardized language for specification of interface provided by an object
- Mappings for IDL exist in all major programming languages
- Used to generate Stub/Skeleton code

```
module HelloApp {
  interface Hello {
  string sayHello();
  oneway void shutdown();
  };
};
```

CORBA – ORB

Object Request Broker (ORB)

- Middleware allowing transparent local and remote invocation
- Handles data serialization/deserialization based on IDL
- Knows location of the actual service implementation
- Is able to handle, e.g., transactions
- General InterORB Protocol – GIOP: Protocol for communications between ORBs



CORBA – Java Implementation Example

```
class HelloImpl extends HelloPOA {
   private ORB orb;

   public void setORB(ORB orb_val) {
      orb = orb_val;
   }

   public void shutdown() {
      orb.shutdown(false);
   }

   // actual function
   public String sayHello() {
      return "\nHello world !!\n";
   }
}
```

5 Remote Execution – Web Services

What is a web service?

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.
>
> — W3C, Web Services Glossary

> 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.
>
> — W3C, Web Services Architecture (2004)

SOAP

- *Simple Object Access Protocol*
- Standard protocol for *web service* communication
- Combo SOAP + WSDL + UDDI
- XML-based, successor of XML-RPC
- In contrast to CORBA:
  - Universal, no language binding (IDL) required
  - XML-based (CORBA protocols binary)
  - Stateless
  - Possibly asynchronous

SOAP

WSDL

- *Web Service Description Language*
- XML-based description of web service interface
- Clients know how to communicate with web service based on WSDL description
  - No generated skeleton or stub needed

UDDI

- Universal Description, Discovery and Integration
- Universal register of WSDL descriptions of SOAP web services
- Simplifies web service discovery

SOAP

SOAP

- Messages consist of:
  - *Envelope* – single per request/response
  - (Optional) *header* – additional information, e.g., timeout, security
  - *Body* – data
  - (Optional) *Fault* – error handling

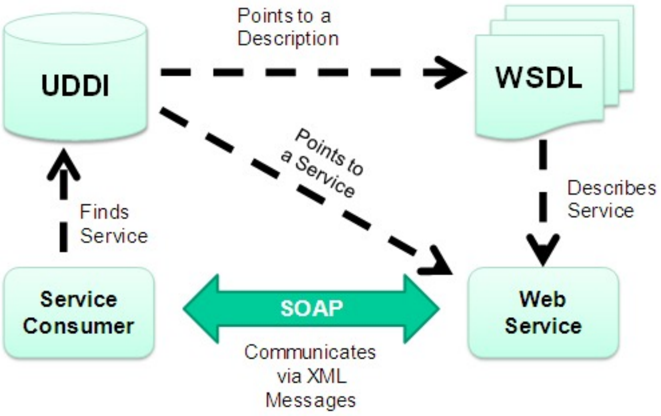


Figure 4: SOAP+WSDL+UDDI. Source: `http://www.wst.univie.ac.at/workgroups/sem-nessi/index.php?t=semanticweb`

- Always sent via HTTP POST
- Annotations allowed generating XML schema automatically
- XML schemas allowed generating client API (e.g. in Java)
- Caveats:
  - VERY complex (and unclear) security model
  - Potentially complex message structure (some information in header insteady of body)
  - Bad opinion about XML, Javascript devs don't like it

SOAP

DEMO

DEMO (opt)

- Create class
- @WebService
- Deploy to Server (Glassfish or Payara), show generated WSDL
- Use NetBeans to quickly generate Client (in Jakarta EE)
- Deploy, run

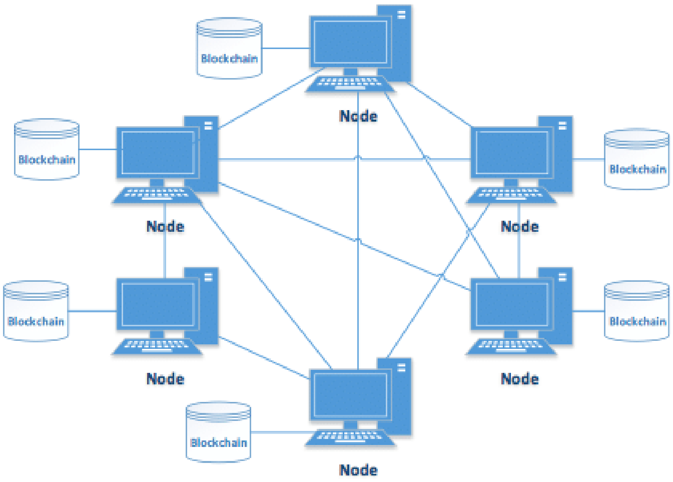


Figure 5: Source: `https://www.researchgate.net/figure/Blockchain-P2P-Network_fig1_320127088`

Peer to Peer (P2P)

- Decentralized architecture where nodes function as servers and clients
- Content distribution, sharing, grid computing
- Types
  - *Unstructured* – no central node, peers discover each other (each peer starts with a few possible connections and builds a list of other peers)
  - *Structured* – network has a topology, more efficient peer discovery
  - *Hybrid* – combination of P2P and client/server – usually server helps clients discover other peers, search etc.

P2P

Architecture

Service Oriented Architecture (SOA)

- System is split into self-contained separate units – *services*
- Services use each other to provide functionality
- Services can be developed separately, use different technologies, be removed or replaced without affecting the system as a whole
- NOT to confuse with Web Services

- Example: SSO, text analysis service

Microservices

- No precise definition exists, for some it is a more advanced (purer) implementation of SOA
- Software units communicating over lightweight mechanisms (HTTP), deployed using automated machinery and DevOps

Communication

SOA – Enterprise Service Bus (ESB)

- ESB is a *middleware*
- Indirection in service communication – decoupling, routing, synchronous or asynchronous communication
- May support multiple protocols – SOAP, REST
- RabbitMQ, Apache Kafka, Apache ActiveMQ

Microservices

- decentralized orchestration, load balancers, cloud tools
  - Each service may have configuration of other possible services it can use
- Or single service registry

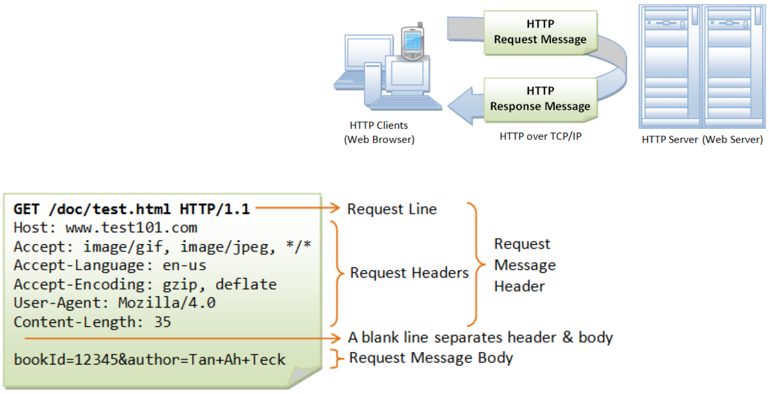
REST

- Next week
- Based on HTTP, let's start with a refresher

6 HTTP

HTTP protocol basics

- HTTP is a client-server application-level protocol
- Typically runs over a TCP/IP connection
- Extensible – e.g., video, image support
- Stateless
- Cacheable
- Requires *reliable* transport protocol – no UDP



```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

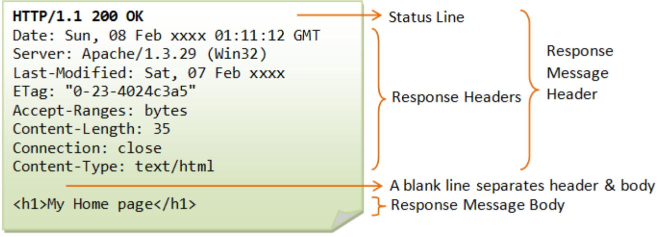
Figure 6: HTTP request example. Source: `https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html`

HTTP Request

- Message header
  - Request line – identifies HTTP method, URI and protocol version
  - Request headers
- Message body

HTTP Response

- Message header
  - Status line – identifies protocol version and response status code
  - Response headers
- Message body



```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```

Figure 7: HTTP request example. Source: `https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html`

	Request	**Response**
Content	• Content-Type • Content-Length • Content-Encoding • Accept	• Content-Type • Content-Length • Content-Encoding
Caching	• If-Modified-Since • If-Match	• Last-Modified • ETag
Miscellaneous	• Cookie • Host • Authorization • User-Agent	• Set-Cookie • Location

HTTP Headers
Typical, often used HTTP headers

HTTP Methods

GET

- Used to retrieve resource at request URI
- Safe and idempotent
- Cacheable
- Can have side effects, but not expected
- Can be conditional or partial (If-Modified-Since, Range)

POST

- Requests server to create new resource from the specified body
- Can be used also to update resources
- Should respond with 201 status and location of newly created resource on success
- Neither safe nor idempotent
- No caching

HTTP Methods

PUT

- Requests server to store the specified entity under the request URI
- Server may possibly create a resource if it does not exist
- Usually used to update resources
- Idempotent, unsafe

DELETE

- Used to ask server to delete resource at the request URI
- Idempotent, unsafe
- Deletion does not have to be immediate

HTTP Response Status Codes

- **1xx** – rarely used
- **2xx** – success
  - 200 OK – requests succeeded, usually contains data
  - 201 Created – returns a *Location* header for new resource
  - 202 Accepted – server received request and started asynchronous processing
  - 204 No Content – request succeeded, nothing to return
- **3xx** – redirection
  - 304 Not Modified – resource not modified, cached version can be used (try `https://javaconferences.org/`)

HTTP Response Status Codes

- **4xx** – client error
  - 400 Bad Request – malformed syntax
  - 401 Unauthorized – authentication required
  - 403 Forbidden – server has understood, but refuses request
  - 404 Not Found – resource not found
  - 405 Method Not Allowed – specified method is not supported
  - 409 Conflict – resource conflicts with client data

  - 415 Unsupported Media Type – server does not support media type

- **5xx** – server error
  - 500 Internal Server Error – server encountered error and failed to process request

400 Bad Request

- Don't use it, if possible, provides no information!
- Real-life xample of Bad Request, found after 2 days of finding the bug:

```
if (!METHODS_TO_IGNORE.contains(rc.getMethod()) && !rc.getHeaders().containsKey("X-
    Requested-By")) {
   throw new BadRequestException();
}
```

7 Conclusions

Conclusions

- Most of today's applications are distributed
  - At least tiered – backend and frontend separate
- Most applications are integrated using web services
- Services allow to build systems from independent modules

Coming Next Week

- HTTP
- Currently most popular Web service architecture – REST

The End

Thank You

Resources

- https://martinfowler.com/bliki/IntegrationDatabase.html
- M. Fowler: Patterns of Enterprise Application Architecture
- http://xmlrpc.scripting.com/spec.html
- http://www.corba.org/
- K. Richta: Standardy pro webové služby WSDL, UDDI
  - https://www.ksi.mff.cuni.cz/~richta/publications/Richta-MD-2003.pdf
- https://www.slideshare.net/PeterREgli/soap-wsdl-uddi
- http://www.aqualab.cs.northwestern.edu/component/attachments/download/228
- https://ifs.host.cs.st-andrews.ac.uk/Books/SE7/Presentations/PDF/ch12.pdf
- https://www.ibm.com/support/knowledgecenter/en/SSMQ79_9.5.1/com.ibm.egl.pg.doc/topics/pegl_serv_overview.html
- https://martinfowler.com/articles/microservices.html