Systems software design

Network communication in complex systems architecture.

Using shared libraries and external components in systems architecture.

Who are we?

Krzysztof Kąkol
Software Developer

Jarosław Świniarski
Software Developer

Presentation based on materials prepared by

Andrzej Ciarkowski, M.Sc., Eng.

outline

- Network communication in complex systems architecture.
  - network programming
  - RPC systems
    - object-oriented bridges
    - WebServices
      - WSDL/SOAP
      - REST
- Using shared libraries and external components in systems architecture.
- Information about the exam

network programming

- the term **network programming** refers to creating applications which **communicate with each other** through computer network
- **network communication** is a specific form of **interprocess communication** (IPC), in which communicating processes are running on different machines
  - we don't use IPC objects whose scope is limited to a single machine (shared memory, semaphores, mutexes…)
  - communication is performed through **network sockets** and **communication protocols**
- the most ubiquitous network communications API are **BSD (POSIX) Sockets** (AKA *Berkeley sockets)*
  - therefore, we commonly refer to network programming as **sockets programming**

network sockets glossary

- **socket** – **endpoint** of IPC based on computer network
- **socket address** – combination of machine's **IP address** and **port number** (service id)
- **socket type**
  - datagram socket (protocol: UDP), type **SOCK_DGRAM**
  - stream-oriented socket (protocol: TCP), type **SOCK_STREAM**
  - raw socket, type **SOCK_RAW** – user is responsible for the implementation of the transport protocol

server and client socket roles

- Stream-oriented sockets (**SOCK_STREAM**) may be used in 2 roles
  - **server sockets** – bound to specific address and port, **listening** for incoming client connections
  - **client sockets** – initiate the connection with listening server socket, don't need to be bound to network address a'priori
- It is not possible to establish session between two client sockets
- In case of connectionless sockets there's no need to create listening sockets and await connection – it's possible to communicate immediately any to any
- The server/client socket division is related to the way they're used – we create them the same way

BSD API – socket()

- int socket(int domain, int type, int protocol);
- **socket()** creates new socket of the specified type, allocates the resources and returns its file descriptor
- parameters
  - **domain** – describes the **protocol family** of the socket:
    - **AF_INET** – IPv4 protocols
    - **AF_INET6** – IPv6 protocols
    - **AF_UNIX** – local sockets (Unix domain sockets – only on Unix)
  - **type** – socket type (**SOCK_DGRAM, SOCK_STREAM, SOCK_RAW, …**)
  - **protocol** – specific transport protocol or 0 eg.
    - **IPPROTO_TCP** – TCP (default for **SOCK_STREAM**)
    - **IPPROTO_UDP** – UDP (default for **SOCK_DGRAM**)

bind()

- int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
- **bind()** binds created socket with local computer address
- newly created socket is not associated with any address; using bind(), we can assign an address, so that we can wait for incoming calls to the specified address/port
- **parameters**
  - **sockfd** - socket file descriptor
  - **addr** – structure describing the address we're binding the socket (IP address + port number). If no IP is given, socket will be bound to **all** machine's IP addresses. If no port number is given, it will be **chosen randomly** by OS)
  - **addrlen** – byte size of addr structure (IPv4/IPv6 compatibility, etc)

listen()

- int listen(int sockfd, int backlog);
- **listen()** - start "listening" for incoming calls to the server socket at the specified address - only for stream-oriented sockets (connection-oriented)
- **parameters**
  - **sockfd** – socket file descriptors
  - **backlog** – length of the queue in which incoming connections are put; we remove the connections from the queue by accepting them with **accept()** call; overflown connections will be automatically rejected

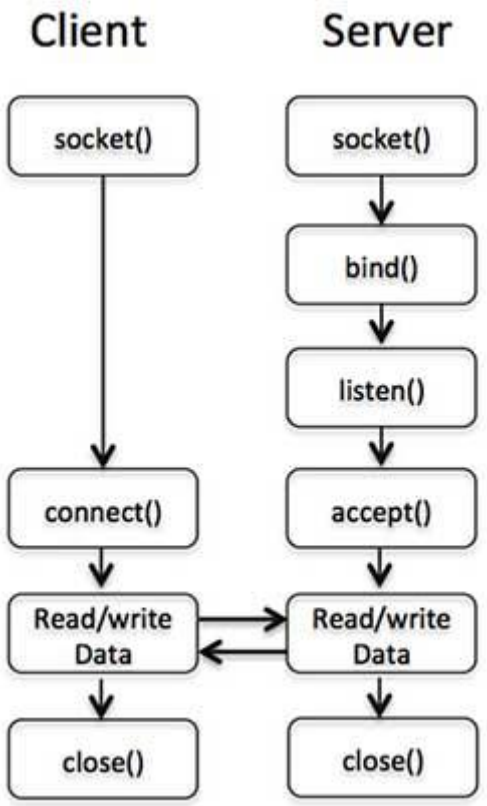
accept()

- int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
- **accept()** accepts an incoming connection on the listening server socket
- the accepted incoming connection creates a new local socket, which is connected to a session with a remote socket
- returns the file descriptor of the new socket
- **parameters**
  - **sockfd** – file descriptor of the listening server socket
  - **cliaddr** – structure filled in with address of the remote socket upon return
  - **addrlen** – byte size of cliaddr structure

connect()

- int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
- **connect()** initiates a new connection of the local client socket to the remote, listening server socket
- may be called for connectionless sockets to define the default target (endpoint) of the communication
- **parameters**
  - **sockfd** – file descriptor of the local client socket being connected to the remote system
  - **serv_addr** – the addres of the remote, listening server socket
  - **addrlen** – byte size of serv_addr

connecting the connection-oriented sockets



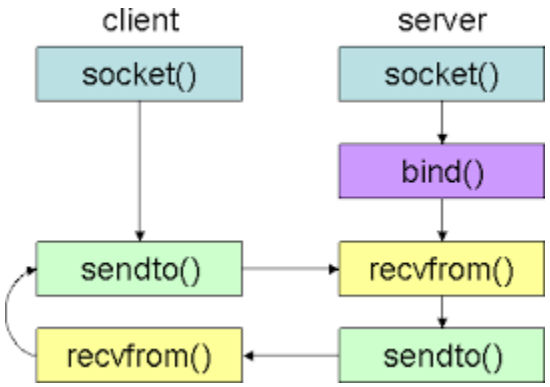
reading/writing the sockets

- depending on the socket type there are two sets of function used for reading/wiritng the socket
  - **connection-oriented** (SOCK_STREAM)
    - read: **ssize_t recv(int** *sockfd*, **void *** *buf*, **size_t** *len*, **int** *flags*);
    - write: **ssize_t send(int** *sockfd*, **const void *** *buf*, **size_t** *len*, **int** *flags*);
  - **connectionless** (SOCK_DGRAM)
    - read: **ssize_t recvfrom(int** *sockfd*, **void *** *buf*, **size_t** *len*, **int** *flags*, **struct sockaddr *** *src_addr*, **socklen_t *** *addrlen*);
    - write: **ssize_t sendto(int** *sockfd*, **const void *** *buf*, **size_t** *len*, **int** *flags*, **const struct sockaddr *** *dest_addr*, **socklen_t** *addrlen*);

reading/writing the sockets

- using connection-oriented API is identical to calling standard file **read()/write()** functions – if **flags** param is 0 (they can be used with sockets too)
- **flags** param allows to set some additional, advanced options which are used very rarely
- connectionless API **recvfrom()/sendto()** allows to set the target address of the datagram or retrieve the source address of the received datagram

using connectionless sockets



network address discovery

- machines in the network are identified by the IP addresses – 32- (IPv4) or 128-bit (IPv6) numbers
- IP addresses are difficult to remember and may change over time, therefore a system of names and name servers was created (**domain name system**, **DNS**), which is a distributed database linking names with IP addresses
- BSD API allows for searching addresses and names of the machines through DNS server queries
- this functionality is called **resolver**
- the basic **resolver** in BSD API is **blocking** – there's no standard, portable way of making non-blocking DNS queries

resolver

- struct hostent *gethostbyname(const char *name);
- **gethostbyname()** returns a list of known addresses for the host with a given name
- **hostent** structure includes fields describing the address type (eg. AF_INET, AF_INET6), host address, its byte size and a **qualified name** linked with this address; link to the next element of the list
- each name may be linked to several addresses of different types and several names (aliases)
- in order to find all addresses/names of the host you need to traverse the list of hostent structures until the next element is empty
- hostent structure is managed by the OS – (is statically allocated) – the **BSD resolver is not reentrant!**

gethostbyaddr()

- struct hostent *gethostbyaddr(const void *addr, int len, int type);
- **gethostbyaddr()** discovers other known addresses and name aliases for the host with specified address and returns their list in an identical way to gethostbyname()
- **parameters**
  - **addr** – address of the host being searched
  - **len** – byte size of the address
  - **type** – type (eg. AF_INET) of the address

POSIX resolver

- **gethostbyname()** and **gethostbyaddr()** are very commonly used, but they are generally unsafe (not reentrant) and **obsolete** and should not be used anymore
- standard POSIX API replaces them with **getaddrinfo()** and **getnameinfo()**, which are more flexible (allow to search of services of specified names) and independent from address domain/protocol
- it’s strongly advised to use new API in new apps (which is more capable but simultaneously less comfortable to use in simple scenarios)

getaddrinfo()

- int getaddrinfo(const char *hostname, const char *service, const struct addrinfo *hints, struct addrinfo **res);
- **getaddrinfo()** allows to find host with the specified name or text-formatted address and the service with the specified name or text-formatted port number
- hint allows to set additional search criteria – the protocol, etc.
- the result is returned as a list of dynamically allocated addrinfo structures, which must be released through **freeaddrinfo()** call

RPC mechanisms

- RPC – **Remote Procedure Call**
- originally the RPC term referred to the protocol created by Sun for development of distributed applications (standard RFC1057), e.g. implementation of NFS filesystem
- nowadays we use the term to describe any technique which allows to remotely invoke services in distributed environment without the need to implement low-level communication details
- in an ideal scenario remote function/procedure invocation through RPC looks to the programmer identical to a local function call and the whole bolierplate code for serialization & transfer of data is called under the hood

typical RPC scenario

- client RPC program calls local **stub** function in his programming language of choice like any other function
- **stub** converts the parameters passed-in on the stack into a message compatible with employed RPC protocol (**parameter serialization**)
- the message is sent to the RPC server offering the specific service
- **RPC server stub** receives the message and deserializes the parameters
- **server stub** invokes the local function with the parameters read from the message and reads the response
- the response is transferred to the client the same way around…

creating client & server stubs

- RPC calls interface is described with a standard IDL language – **interface description language**
- based on the IDL description, the **code generator** of the RPC system creates the **binding** between the stubs and RPC library in a given programming language, providing the necessary serialization/deserialization (marshalling) code
- stubs are being compiled and linked to the client & server code
- the implementation of the stubs on the client & server sides may use different programming languages as long as common RPC protocol is maintained (**heterogenous**)

IDL example

```
interface Hello
{
    string sayHello(in string name);
}
```

C++ object reference – client stub invocation

```
class HelloRef: public Hello
{
public:
    virtual std::string sayHello(const std::string& name);
};

int main(int argc, char** argv)
{
    HelloRef hello;
    std::cout << hello.sayHello(argv[1]) << std::endl;
    return 0;
}
```

Java server stub implementation

```
class HelloImpl extends Hello
{
    String sayHello(String name)
    {
        return "Hello, " + name + ", how are you?";
    }
}
```

object-oriented interfaces

- modern RPC implementations don’t model single functions but rather object interfaces
- object-oriented design allows for creating “remote”objects storing the state and eases the hiding of RPC system’s implementation details

Web Services

- **Web Services** is an RPC approach based on using standard Internet protocols and data formats (HTTP, XML, JSON)
- a great feature is the ease of developing and debugging – using known, commonly used formats and tooling
  - no need to maintain full, complex server infrastructure like in the case of CORBA – WebServices run on standard WWW servers
  - lower chance that HTTP calls would be blocked by the firewalls, ability to make use of standard Web Proxy servers
- similarly to “traditional” RPC there are tools which allow for generating stub/skeleton code from IDL or special WSDL language ((Web Service Definition Language – an XML dialect)

Web Services approaches

- formal
  - WSDL interface description
  - SOAP communication protocol
  - support for tooling, generators – Java, .NET
  - **excess of form over substance** – invoking our „sayHello()” to transfer a few characters requires several thousand characters of XML data
- loose, ad hoc – „everything that looks and works like web service is web servce”
  - XML-RPC – simple fragments of XML sent through HTTP
  - JSON-RPC – like above, but the data in JSON format
  - RESTful approach
  - **interoperability most of the time is maintained only between the stubs generated by the same library**
  - support from scripting languages – Python, PHP, JavaScript

SOAP example

Request

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-
encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Response

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-
encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

REST concept

- Using different HTTP methods to perform different actions
  - **GET http://example.com/books/123**
    gets data from book collection about book with id 123
  - **POST http://example.com/books**
    creates new book entity, new book data is located in POST body
  - **PUT http://example.com/books/123**
    change data about book with id 123, data located in PUT body
  - **PUT http://example.com/books/123**
    removes data about book with id 123 from collection

REST example

GET http://example.com/books/123

```
{
   ”id” : 123,
   ”title" : „Harry Plumber",
   ”noOfPages” : 500,
   „pages” : [
       { //page data },
       { //page data },
       { //page data },
   ]
}
```

PUT http://example.com/books/123

```
{
   ”id” : 123,
   ”title" : „Harry Potter",
}
```

outline

- Network communication in complex systems architecture.
- Using shared libraries and external components in systems architecture.
  - programming libraries
    - static
    - shared
  - dynamic linking
  - library interface
  - software plugins
  - backward compatibility
- Information about the exam

programming libraries

- **library** – collection of resources and procedures/function used by computer programs through well-defined interface
  - code provided through the library may be easily used by multiple unrelated programs – **code reuse**
  - program needs not to know the implementation of the library function – it relies solely on their interface
  - services delivered by the library are related to the common set of subjects

library types

- libraries may be divided according to their **linking** type – during which build/execution phase the library is being linked to the executable program
- **static** libraries – collections of compiled object code file, which are linked directly to the executable file during its build; every executable file receives private **copy** of the library code
- **shared** libraries (dynamically linked) – library code is not copied to executable file, but the separate library file is referenced; the same library may be **shared** simultaneously among many programs

library comparison

static

- copying of code
- any change to library requires recompilation/linking of every executable using it
- the executable program consists of only single file (easy to distribute)
- program is **self-contained**
- no additional linking phase is required during the execution – no startup delay

shared

- referencing shared code
- it's possible to change the library through replacement of its file, provided the **binary interface** and **backward compatibility** are maintained
- executable program requires the additional library files to work
- during the program execution there's the dynamic linking phase which may have substantial runtime cost
- changes/fixes to the library files don't require to install new program versions
- lower consumption of disk space (only 1 library file) and RAM (library image is shared among processes)

dynamic linking

- executable program stores the name of the library and the names of the **symbols exported by the library** or **indices to the library symbol table**
- after starting the process, the dynamic linker of OS is used to:
  - locate the library file
  - load library image to process'address space
  - perform necessary **symbol relocations**
  - store the references to the symbols from the library in appropriate slots of program's symbol table
  - running the library startup code (eg. DllMain(), dlinit(), _declspec_(constructor))

runtime loading of the libraries

- dynamic linking needs not to be performed during program startup – it's possible at any moment of its execution
- **run-time linking**
  - implicit – performed automatically during the first reference of any symbol from the delay-loaded library (**lazy binding**) – shortens the startup time
  - explicit – program code explicitly calls the dynamic linker functions to load the library and access its named symbols – the library needs not to be accessible during program startup – its functionality is **optional** – the "plugins"

runtime loading of libraries

POSIX

```
void* lib = dlopen("library.so", RTLD_LAZY);
if (lib == NULL) {
    // report error ...
} else {
    // use the result in a call to dlsym, find symbol in the lib
    void* symbol = dlsym(lib, "symbol");

    typedef int (*function_t)(int param);
    function_t fun = (function_t)symbol;
    int result = fun(5);
    dlclose(lib);
}
```

Windows

```
HMODULE hLib = LoadLibrary("library.dll");
if (hLib == NULL) {
   // report error ...
} else {
   // use the result in a call to GetProcAddress, find symbol in the lib
    void* symbol = GetProcAddress("symbol");

    typedef int (* function_t)(int param);
    function_t fun = (function_t)symbol;
    int result = fun(5);
    FreeLibrary(hLib)
}
```

runtime loading of libraries

- the type of the symbol (function prototype, its linkage, parameters & result types) must be known a'priori – malformed prototype will result in destruction of stack and program crash
- the name of the symbol must be known a'priori – in case of C++ the **decorated name** (unportable, that's why usually we use "**extern C**" linkage)
- symbol name may be the exported function or global variable

library interface

- **library interface** is a list of all **exported** symbols and their prototypes
- usually the library interface is provided in the form of the header (.h) files accompanying the library, containing the declarations of the exported symbols

program plugins

- „plugin” is a special type of shared library which is designed to be loaded during the program runtime
- plugins provide unified interface based on symbols with known names and/or classes with pure virtual interface
- plugins are required to have compatible **binary interface**
  - implementation hiding
  - using **opaque** objects
  - access to the state and behaviours of the object solely through the function calls, data encapsulation

program plugins

```cpp
#include <dsp++/export.h>

class plugin_interface
{
public:
    virtual ~plugin_interface() = 0;
    virtual initialize() = 0;
    virtual do_something() = 0;
    virtual dispose() = 0;
};

typedef plugin_interface* (*plugin_create_fun)();
typedef void (*plugin_destroy_fun)(plugin_interface*);

extern "C" DSPXX_API plugin_interface* plugin_create();
extern "C" DSPXX_API void plugin_destroy(plugin_interface*);
```

```cpp
#include "plugin_interface.h"

...

void* plugin = dlopen("plugin", 0);
plugin_create_fun create = (plugin_create_fun) dlsym(plugin, "plugin_create");
plugin_destroy_fun destroy = (plugin_destroy_fun) dlsym(plugin, "plugin_destroy");

plugin_interface* pi = create();

pi->initialize();
pi->do_something();
pi->dispose();

destroy(pi);

dlclose(plugin);
```

backward compatibility

- typical library versioning assumptions
  - code using earlier version of the library should work with the next library version, provided the **major version number** didn't change
  - **breaking changes** should result in the change of **major version number**
  - typical version number syntax:
    major_number.minor_number[.**patch_number**[.**build_number**]]
  - **major_number** – number of **binarnego interface** version; changed on major change of functionality or library design
  - **minor_number** – changed on functionality enhancements which don't break backward compatibility
  - **patch_number** – bug fixes without any change to the library interface
  - **build_number** – number of the compilation, used for tracing the exact day the bug was introduced

information about the exam

- Next week we've got the EXAM... :)
- But don't worry, it will not be difficult.
- The questions will be rather descriptive, however not very detailed. There will also be a couple of single choice questions.
- Questions will cover the full range of discussed subjects – but will not go beyond the contents of the presentations!

question samples

1. What is DDD (please explain the acronym and shortly describe the purpose of DDD)?
2. What is the name of the list of tasks in Scrum prepared by the product owner?
3. Which of the following roles is a part of the scrum team:
   a) product owner
   b) stakeholders
   c) company manager
   d) project manager
4. What types of build configurations will be usually created by modern IDEs?
5. What is „multithreading”?

exam rules

1. There will be around 20 questions.
2. Exam will take around 30-40 mins – so not too much time for answering.
3. Every question answered correctly will be awarded with 1 point (single-choice) or 2 points (descriptive).
4. Passing score is 50%.

Questions?

Krzysiek kkakol@pgs-soft.com
Jarek jswiniarski@pgs-soft.com

www.pgs-soft.com