Status Report: NET

Abhinav Gauniyal
Jitender Galav
Tushar Khurana
Mohd. Harris

Overview

Description

Open Source, modern C++ library for building network-based applications.

Goals

- Consistent with berkeley sockets api
- Provide high level abstractions
- Modular design with few external dependencies
- Policy based design
- Compile time safety
- Easy to extend to different platforms

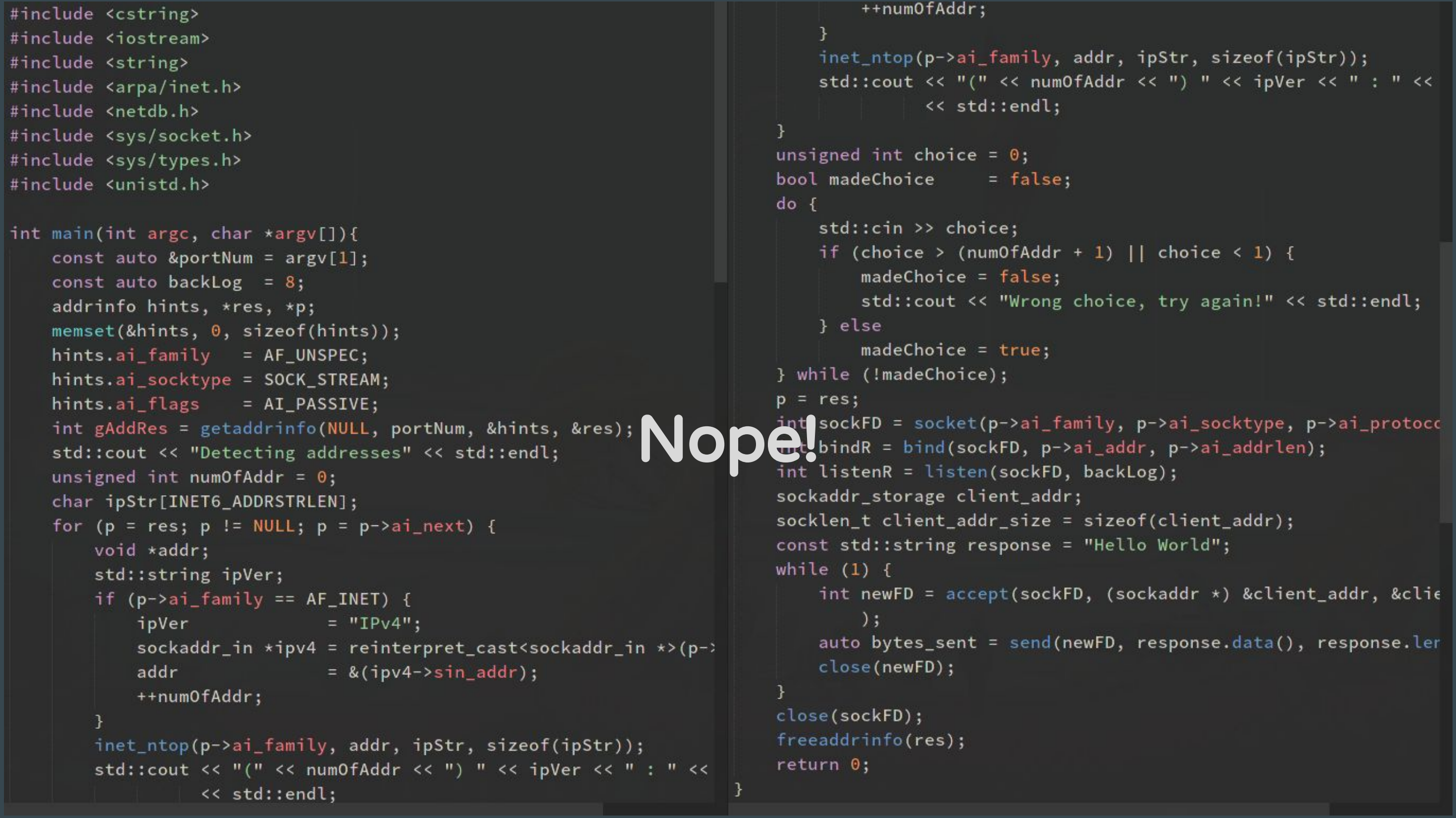
Berkeley sockets

Berkeley sockets is an application programming interface (API) for Internet sockets and Unix domain sockets, used for inter-process communication (IPC). It originated with the 4.2BSD Unix released in 1983.

Berkeley sockets evolved with little modification from a de facto standard into a component of the POSIX specification. Therefore, the term POSIX sockets is essentially synonymous with Berkeley sockets. They are also known as BSD sockets, acknowledging the first implementation in the Berkeley Software Distribution.

Since most Unix derivatives follow POSIX standard, implementing a library in conformance to Berkeley Sockets API is the only sane choice. Fortunately, Winsock/2 also is based upon them.

High Level Network Programming



```cpp
#include <cstring>
#include <iostream>
#include <string>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[]){
    const auto &portNum = argv[1];
    const auto backLog  = 8;
    addrinfo hints, *res, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;
    int gAddRes = getaddrinfo(NULL, portNum, &hints, &res);
    std::cout << "Detecting addresses" << std::endl;
    unsigned int numOfAddr = 0;
    char ipStr[INET6_ADDRSTRLEN];
    for (p = res; p != NULL; p = p->ai_next) {
        void *addr;
        std::string ipVer;
        if (p->ai_family == AF_INET) {
            ipVer             = "IPv4";
            sockaddr_in *ipv4 = reinterpret_cast<sockaddr_in *>(p->
            addr              = &(ipv4->sin_addr);
            ++numOfAddr;
        }
        inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
        std::cout << "(" << numOfAddr << ") " << ipVer << " : " <<
                  << std::endl;
```

```cpp
            ++numOfAddr;
        }
        inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
        std::cout << "(" << numOfAddr << ") " << ipVer << " : " <<
                  << std::endl;
    }
    unsigned int choice = 0;
    bool madeChoice     = false;
    do {
        std::cin >> choice;
        if (choice > (numOfAddr + 1) || choice < 1) {
            madeChoice = false;
            std::cout << "Wrong choice, try again!" << std::endl;
        } else
            madeChoice = true;
    } while (!madeChoice);
    p = res;
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protoco
    int bindR = bind(sockFD, p->ai_addr, p->ai_addrlen);
    int listenR = listen(sockFD, backLog);
    sockaddr_storage client_addr;
    socklen_t client_addr_size = sizeof(client_addr);
    const std::string response = "Hello World";
    while (1) {
        int newFD = accept(sockFD, (sockaddr *) &client_addr, &clie
            );
        auto bytes_sent = send(newFD, response.data(), response.ler
        close(newFD);
    }
    close(sockFD);
    freeaddrinfo(res);
    return 0;
}
```

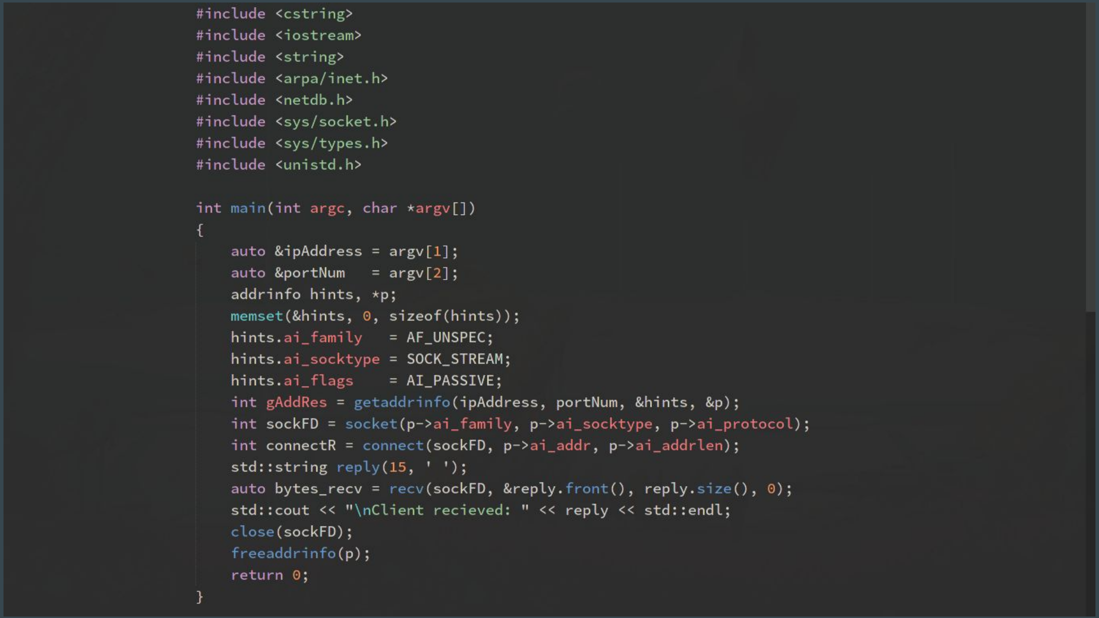
High Level Network Programming

Previous slide contained a screenshot of simple TCP server program with

- No error handling
- No support for concurrent clients
- No support for multiple processes
- No support for multithreaded design
- A potential memory leak
- Only TCP + IPv4 agnostic
- Modularity might be a joke here
- And definitely not cross-platform

And looking at the LOC one can imagine the amount of time spent on refactoring, debugging and maintainability would increase exponentially with the size of program.

High Level Network Programming

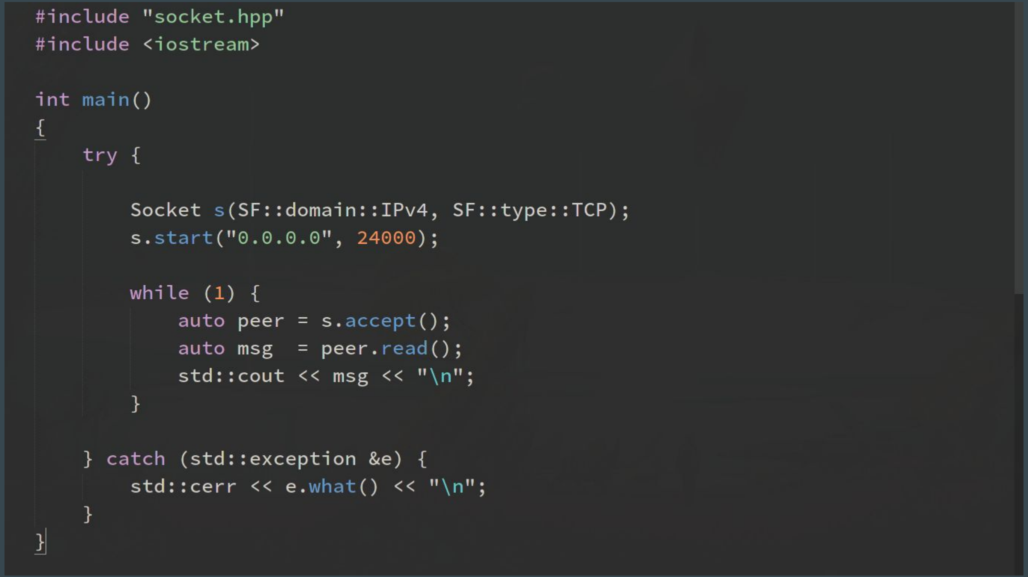


```
#include <cstring>
#include <iostream>
#include <string>
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    auto &ipAddress = argv[1];
    auto &portNum   = argv[2];
    addrinfo hints, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;
    int gAddRes = getaddrinfo(ipAddress, portNum, &hints, &p);
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    int connectR = connect(sockFD, p->ai_addr, p->ai_addrlen);
    std::string reply(15, ' ');
    auto bytes_recv = recv(sockFD, &reply.front(), reply.size(), 0);
    std::cout << "\nClient recieved: " << reply << std::endl;
    close(sockFD);
    freeaddrinfo(p);
    return 0;
}
```

The corresponding client program for same TCP+IPv4 server we just saw comparatively looks small enough, but again has no error handling built in and is a bit verbose too.

Solution and stuff

High Level Network Programming

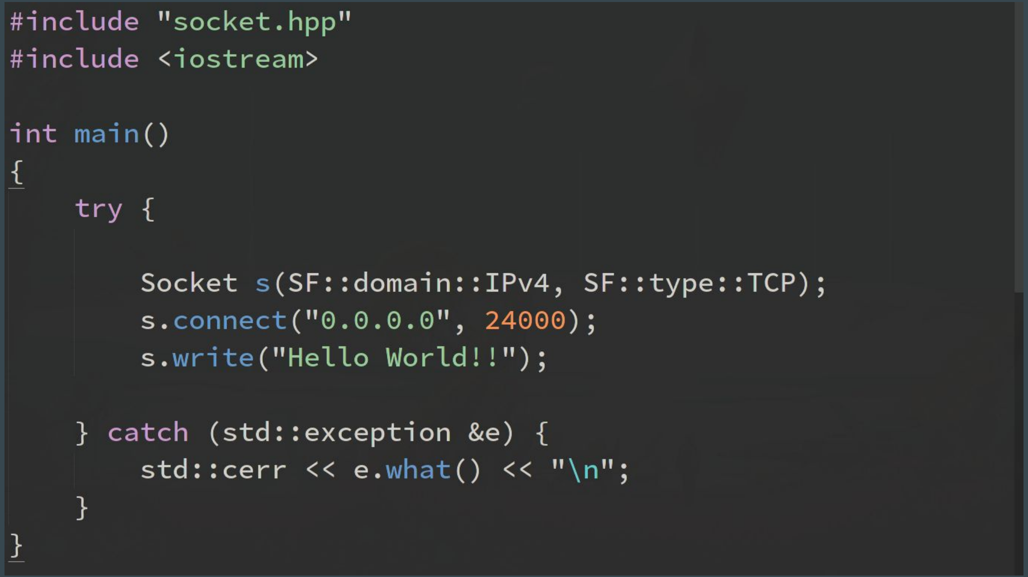


```
#include "socket.hpp"
#include <iostream>

int main()
{
    try {

        Socket s(SF::domain::IPv4, SF::type::TCP);
        s.start("0.0.0.0", 24000);

        while (1) {
            auto peer = s.accept();
            auto msg  = peer.read();
            std::cout << msg << "\n";
        }

    } catch (std::exception &e) {
        std::cerr << e.what() << "\n";
    }
}
```

That’s it.

The following program creates a TCP+IPv4 server, starts it on specified port, reads incoming messages, and reports them on standard output. Errors are handled gracefully too.

High Level Network Programming



```
#include "socket.hpp"
#include <iostream>

int main()
{
    try {

        Socket s(SF::domain::IPv4, SF::type::TCP);
        s.connect("0.0.0.0", 24000);
        s.write("Hello World!!");

    } catch (std::exception &e) {
        std::cerr << e.what() << "\n";
    }
}
```

Here's your corresponding client program that connects to the server on specified port, sends a message and exits. Effectively, a 3 lines client program.

Modularity

Previous slides contained screenshots which displayed a *Socket* object which was used to communicate as a server and client. It might've seemed that Socket resembles a server/client object but it isn't. The carefully crafted API provides all underlying system calls encapsulated with the *Socket* object hence making it a building block. Such design makes it possible to:

- Use as or Build framework for TCP for both IPv4 & IPv6 families
- Use as or Build framework for UDP for both IPv4 & IPv6 families
- Use as or Build framework for UNIX sockets for both IPv4 & IPv6 families
- Similarly for other socket families including but not limited to AF_NETLINK, AF_X25, AF_PACKET

The NET library itself provides framework for building TCP/UDP programs across both IPv4/6 domains, thus making the library modular.

Concurrent Connections

All the examples shown yet are of single threaded iterative server which is rarely used in practice(at least for TCP). The lowest entity *Socket* has no understanding of concurrent connections hence the TCP/UDP frameworks will provide some strategies/policies to launch server/client programs which have concurrency capabilities.

They'll be based upon:

- Forking an entirely new process for each new connection
- Launching a new thread for each new connection
- Select system call with non-blocking *Socket* calls
- Thread pool, poll/Epoll, async i/o are also in consideration

Schedule

03.02.2017

Project requirements, design analysis and schedule decided.

15.02.2017

Initial project presented, review gathered, amendments made

24.02.2017

Socket entity completed, fully conformant to Berkeley Socket API.

05.03.2017

Frameworks completed, Extensive testing performed, prepare for release.

XX.XX.2017

Benchmarks completed, released v1.0, published report.

Progress

1. Partial Socket Implementation Completed. Error handling and some system calls remaining to be implemented.
2. Frameworks with policy design pattern to be implemented.
3. Maintenance/Bugs + Benchmarks to be conducted

Thank You